Master's Thesis

A Distributed Forest of Octrees Datastructure for Particle Wigner Simulations

Ausgeführt am
Institut für Mikroelektronik

der Technischen Universität Wien

unter Anleitung von

Associate Prof. Dipl.-Ing. Dr.techn. Josef Weinbub

Associate Prof. Dr.techn. Lado Filipovic

durch

Ferdinand Maximilian Mond

Matrikelnummer 11945397

Studienkennzahl 066 646

Wien, April 2024

Abstract

This thesis explores the performance characteristics and potential bottlenecks of using the p4est library for mesh management in a Monte Carlo particle Wigner simulator. The goal is to develop a benchmark library which encompasses all features needed for a Monte Carlo particle Wigner simulation, while working as a minimal working example of these features to determine where bottlenecks lie. This benchmark library is then run on the VSC5, where the runtime is used as the performance measurement and recorded for every part of the benchmark. The results of this benchmark show that p4est is a strong candidate for a mesh library in the next iteration of the ViennaWD [1].

The benchmark focuses on the generation and deletion of particles, the communication between processors, and the connection between particles and cells in the mesh, which is the most compute intensive part of the benchmark. There are a number of design choices made in the benchmarking library relating to data storage, communication strategy, and particle generation events. A two-dimensional array is used to store all particle data, either as an Array of Structs (AoS), each struct representing a single particle, or a Struct of Arrays (SoA) where each array represents a variable, stored for each particle. Communication is introduced both in a one-shot method as well as a round-based method. Each individual step of the benchmark is then timed to find its performance characteristics with a different number of particles, mesh elements, and processes in the benchmark, expanding to multiple nodes of the VSC cluster for large simulations.

Performance scaling with increasing processor count is found to be linear when in a load balanced state. Scaling with increasing particle count is found to be roughly linear, although a communication bottleneck limits the quality of data for this benchmark. Performance scaling with increasing number of mesh cells is found to be extremely strong, decreasing runtime by two orders of magnitude with eight orders of magnitude increases in quadrant count.

The limiting factors of performance are found to be in the communication throughput of the VSC and general performance when in a very unbalanced load state, with a specific bottleneck in communication when increasing communication load above a certain threshold. The memory footprint of the benchmarking library in its particle storage is found to be without any significant overhead, while the memory footprint of the quadrant storage managed by p4est is found to scale linearly with the number of quadrants. Some further optimizations, such as compiler flags or different MPI libraries, were not explored in this thesis.

P4est is found to be a strong candidate for a mesh management library in the context of a Monte Carlo particle Wigner simulation, supporting the particle-cell relationships that are required for the simulation in a highly scalable way.

[1] https://viennawd.sourceforge.net/

Kurzfassung

Diese Arbeit untersucht die Leistungsmerkmale und potenziellen Engpässe bei der Verwendung der p4est-Bibliothek für das Mesh-Management in einem Monte-Carlo-Partikel-Wigner-Simulator. Das Ziel ist die Entwicklung einer Benchmark-Bibliothek, die alle für eine Monte-Carlo-Partikel-Wigner-Simulation benötigten Funktionen umfasst, während sie als minimales Arbeitsbeispiel dieser Funktionen dient, um festzustellen, wo Engpässe liegen. Diese Benchmark-Bibliothek wird dann auf dem VSC5 ausgeführt, wobei die Laufzeit als Leistungsmessung verwendet und für jeden Teil des Benchmarks aufgezeichnet wird. Die Ergebnisse dieses Benchmarks zeigen, dass p4est ein starker Kandidat für eine Mesh-Bibliothek in der nächsten Iteration des ViennaWD [2] ist.

Der Benchmark konzentriert sich auf die Generierung und Löschung von Partikeln, die Kommunikation zwischen Prozessoren und die Verbindung zwischen Partikeln und Zellen im Mesh, welcher der rechenintensivste Teil des Benchmarks ist. Es wurden eine Reihe von Designentscheidungen in der Benchmark-Bibliothek getroffen, die sich auf die Datenspeicherung, Kommunikationsstrategie und Partikelgenerierungsereignisse beziehen. Ein zweidimensionales Array wird verwendet, um alle Partikeldaten zu speichern, entweder als ein Array von Strukturen (AoS), wobei jede Struktur ein einzelnes Partikel repräsentiert, oder eine Struktur von Arrays (SoA), wobei jedes Array eine Variable repräsentiert, die für jedes Partikel gespeichert wird. Die Kommunikation wird sowohl in einer Einmalschuss-Methode als auch in einer rundenbasierten Methode eingeführt. Jeder einzelne Schritt des Benchmarks wird dann zeitlich gemessen, um seine Leistungsmerkmale mit einer unterschiedlichen Anzahl von Partikeln, Maschenelementen und Prozessen im Benchmark zu finden, erweitert auf mehrere Knoten des VSC-Clusters für große Simulationen.

Die Leistungsskalierung mit zunehmender Prozessoranzahl zeigt sich als linear, wenn sie sich in einem ausgewogenen Lastzustand befindet. Die Skalierung mit zunehmender Partikelanzahl wird als annähernd linear gefunden, obwohl ein Kommunikationsengpass die Qualität der Daten für diesen Benchmark einschränkt. Die Leistungsskalierung mit zunehmender Anzahl von Maschenzellen wird als extrem stark befunden, wobei die Laufzeit um zwei Größenordnungen mit acht Größenordnungen Erhöhungen in der Quadrantenzahl verringert wird.

Die begrenzenden Faktoren der Leistung werden in der Kommunikationsdurchsatz des VSC und der allgemeinen Leistung in einem sehr unausgewogenen Lastzustand gefunden, mit einem spezifischen Engpass in der Kommunikation, wenn die Kommunikationslast über einen bestimmten Schwellenwert erhöht wird. Der Speicherbedarf der Benchmark-Bibliothek in ihrer Partikelspeicherung wird als ohne signifikanten Overhead gefunden, während der Speicherbedarf der Quadrantenspeicherung, die von p4est verwaltet wird, linear mit der Anzahl der Quadranten skaliert. Einige weitere Optimierungen, wie Compiler-Flags oder unterschiedliche MPI-Bibliotheken, wurden in dieser Arbeit nicht erforscht.

P4est wird als starker Kandidat für eine Mesh-Management-Bibliothek im Kontext einer Monte-Carlo-Partikel-Wigner-Simulation bewertet, der die für die Simulation erforderlichen Partikel-Zell-Beziehungen auf eine hoch skalierbare Weise unterstützt.

[2] https://viennawd.sourceforge.net/

Acknowledgements

I would like to express my sincerest gratitude to Prof. Josef Weinbub for his invaluable guidance, patience, and expertise in supervising the research and writing of this thesis. His insights and feedback were crucial to the development and direction of this work, and his encouragement motivated me to persevere through the challenges encountered along the way. I am eternally grateful for the opportunity he afforded me in gaining access to the largest supercomputer in Austria, the VSC, which is a great honor as a masters student. I am also profoundly thankful to Prof. Lado Filipovic, who graciously took over the supervision of my thesis after Prof. Weinbub left the university. Prof. Filipovic's support and understanding were instrumental in the completion of this project. His willingness to step in at a crucial time and provide the necessary guidance and advice was greatly appreciated.

Moreover, I am grateful to Mihail Nedjalkov, Mauro Ballicchia, Clemens Etl, and Paul Manstetten for our enriching weekly group meetings and additional support. These discussions offered me a greater insight into the academic realm, covering topics from writing scientific papers to participating in international conferences and seminar presentations at the Institute of Microelectronics. These sessions notably broadened my understanding and appreciation for the rigors of research, significantly contributing to my academic growth.

I cannot forget to acknowledge the unwavering support and love of my family. My father Mathias, my mother Ingeborg, and my brother Moritz have been my pillars of strength throughout this journey. Their help with the proofreading, their general support during the writing process, and their financial backing were indispensable. Their belief in my abilities and their encouragement to pursue my goals have been the foundation upon which this achievement was built.

This thesis would not have been possible without the contributions of each of these individuals. I am eternally grateful for their support, encouragement, and belief in my capabilities.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass die vorliegende Arbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen von mir selbstständig erstellt wurde. Alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, sind in dieser Arbeit genannt und aufgelistet. Die aus den Quellen wörtlich entnommenen Stellen, sind als solche kenntlich gemacht. Das Thema dieser Arbeit wurde von mir bisher weder im In- noch Ausland einem_r Beurteiler_in zur Begutachtung in irgendeiner Form als Prüfungsarbeit vorgelegt. Diese Arbeit stimmt mit der von den Begutachter_innen beurteilten Arbeit überein. Ich nehme zur Kenntnis, dass die vorgelegte Arbeit mit geeigneten und dem derzeitigen Stand der Technik entsprechenden Mitteln (Plagiat-Erkennungssoftware) elektronisch-technisch überprüft wird. Dies stellt einerseits sicher, dass bei der Erstellung der vorgelegten Arbeit die hohen Qualitätsvorgaben im Rahmen der geltenden Regeln zur Sicherung guter wissenschaftlicher Praxis "Code of Conduct" an der TU Wien eingehalten wurden. Zum anderen werden durch einen Abgleich mit anderen studentischen Abschlussarbeiten Verletzungen meines persönlichen Urheberrechts vermieden.

Ort, Datum

Name

Contents

1 Introduction

The goal of this thesis is to evaluate the performance of the so-called *forest-of-octrees* data structure as provided by the open source C-library p4est [3] for potential utilization as a fundamental mesh data structure for the open source Monte Carlo particle Wigner simulator ViennaWD [4]. ViennaWD currently only supports regular two-dimensional (2D) Euclidian grids, drastically limiting modeling capabilities, as realistic and practically more relevant simulation scenarios consist of intricate three-dimensional (3D) geometries. Simultaneously, the Wigner-specific particle solver requires certain high performance particle-cell relationships and mesh element topologies for efficient numerical implementation. Therefore, the forest-of-octrees data structure was identified as a prime candidate, as it supports cuboid mesh element topologies, dynamic mesh adaptation, and arbitrary 3D geometries. In addition, p4est was already applied in a plethora of very large-scale computational problems, indicating the, in principle, high scalability of the data structure and its implementation in p4est, which in turn holds great promise for future highly parallel implementations of ViennaWD.

However, p4est has not been evaluated yet regarding its suitability as a key mesh data structure for a Wigner particle simulator. This requires careful evaluation and such is the focus of this work. More concretely, a benchmark library was developed which allows for a tailored performance evaluation of p4est, using a minimal working example which includes key algorithmic aspects representative of the particle solver algorithm used in ViennaWD. Rigorous benchmarks are presented and potential bottlenecks identified, resulting in specific recommendations for further development. Ultimately, the goal is to use this benchmark library and the findings to explore the suitability of p4est as the central data structure for developing the next generation of ViennaWD.

For a particle Wigner simulation performance is critical as it determines the fidelity at which it can be executed in the same timeframe, both in space as well as temporal discretization, the speed at which new simulations can be completed, the cost of these simulations, or a combination of all three. Therefore, finding any bottlenecks which may limit performance before implementing the particle Wigner simulator itself can help to avoid pitfalls in later development and improve the results which are achievable in the simulation.

In the remainder of this Chapter, a short overview over the field of computational electronics and particle Wigner simulation is provided and some terminology and the overall objectives are laid out. The p4est library is introduced in Chapter 2 along with other relevant background information. The implementation of the benchmark library is then detailed in Chapter 3 before analysing the performance of the benchmark in a variety of scenarios in Chapter 4. Finally, Chapter 5 provides an analysis of the

[3]https://www.p4est.org/index.html
[4]https://viennawd.sourceforge.net/

memory footprint of the benchmark and p4est followed by the conclusion and outlook of the thesis in Chapter 6.

1.1 Computational Electronics

The objective of modelling and simulation in computational electronics is to create and apply methodologies that are accurate enough to simulate essential physical phenomena while minimizing computational demands so that results may be obtained in a reasonable timeframe [1]. These models are then applied to stimulate the development of new innovations and technologies in electronic devices and assist in their design process. It combines sophisticated computational techniques to analyze behaviours like quantum effects and electromagnetic fields at the nanometer scale. The ultimate aim of this field is to bridge the gap between theoretical research and practical application, providing a robust platform for the exploration of new materials, devices, and circuit designs. By accurately predicting how these components behave under a wide range of conditions, computational electronics facilitates the rapid prototyping and testing of innovative concepts. This not only accelerates the pace of technological development but also significantly reduces the cost and time associated with traditional experimental approaches.

The most relevant part of the simulations within computational electronics are the quantum mechanical effects, as these require novel methods and high computational load to solve in comparison to the classical physics effects. The transistors within a modern CPU have shrunk to such a small size that quantum tunnelling and other quantum-mechanical effects have significant impact on the functionality of the transistor and therefore need to be accurately simulated [2].

While the most common description of quantum mechanics is that developed by Schrödinger [3], it is not the only possible description of quantum mechanics. Other formulations were developed which are mathematically equivalent but provide other possible benefits such as the one by Wigner [4] or Feynman [5]. The Schrödinger formulation relies on a wave function to describe particles while the Wigner formulation introduces quasi-distribution functions and the Feynman formulation introduces path integrals as equivalent formulations. These formulations are very difficult to solve analytically [6], however the Monte Carlo method can be used to solve them with high accuracy and relatively low numerical complexity.

1.2 Particle Wigner Simulation

The particle Wigner simulation implemented in ViennaWD applies a Monte Carlo method. The Monte Carlo method relies on random sampling and was developed to solve mathematical problems that may be deterministic in principle but are too complex for analytical solutions. There are two applications of the Monte Carlo method, the Monte Carlo simulation algorithms and the Monte Carlo numerical algorithms. Monte Carlo simulation algorithms are used for simulating real life processes and phenomena by following their respective physical, chemical or biological processes. Monte Carlo numerical algorithms on the other hand are usually used to solve deterministic problems by modelling random variables or fields. This is done by constructing an artificial random process and proving that the mathematical expectation of this process is equal to the result of the problem being simulated [7]. This statistical approach is particularly effective for exploring quantum mechanics, where analytical solutions of problems are difficult if not impossible to find in realistic situations.

The particle Wigner simulator ViennaWD uses a sophisticated computational technique that merges the principles of the Monte Carlo method with the quantum mechanical Wigner function for simulating quantum phenomena in semiconductor devices. It effectively works by performing two simulations in parallel which are interlaced: the particle simulation which tracks how particles move through the domain and the field simulation which simulates how an electromagnetic field behaves along the discretized domain. Among the developed particle Wigner methods are the ones applying the concepts of particle quantum affinity [8] and signed particles [9], respectively. The latter is used by Vienna WD and is thus at the core of this thesis.

In practice, the Monte Carlo Particle Wigner method simulates the trajectories of individual particles, such as electrons, through random sampling while accounting for quantum mechanical effects through the Wigner function. Most successful device simulation has been based on solving the Boltzmann transport equation, however this method does not allow quantum mechanical interference phenomena to be included. The Wigner function on the other hand allows these quantum mechanical effects to be incorporated without breaking the uncertainty principle [10], which in turn enables the detailed analysis of quantum phenomena, including tunneling, superposition, and entanglement, which are critical for understanding the behavior of nanoscale semiconductor devices. The method's strength lies in its ability to model these complex quantum interactions with high accuracy, while the Monte Carlo approach facilitates the reduction of the computational complexity inherent in quantum systems. By bridging the gap between quantum physics and practical simulation, the Monte Carlo Particle Wigner method offers a powerful tool for advancing our understanding and capability to engineer the next generation of quantum-influenced semiconductor technologies.

The flowchart of the particle Wigner simulator ViennaWD is shown schematically in figure 1. The ViennaWD simulator first initializes the domain decomposition and particle setup among others based on the input parameters like geometry and potential. Then there are four steps that are taken for each time step in the ViennaWD simulator: evolution, growth prediction, annihilation, and particle transfer. The evolution step deals with all particles existing at the beginning of the time step by simulating their movement, and then generating new particles through either phonon scattering or particle generation. This feeds into the growth prediction step, where for each process locally a prediction is made to determine if in the next time step the number of particles will go above a certain threshold. If any process determines this is the case, during the annihilation step a global annihilation flag is broadcast and particles are destroyed on every process. The particle transfer step ensures that particles which have moved are transferred between processes if required. The simulator loops through these four steps for every time step until it is determined to be complete, at which point data is collected, merged and post-processed for output to a file.

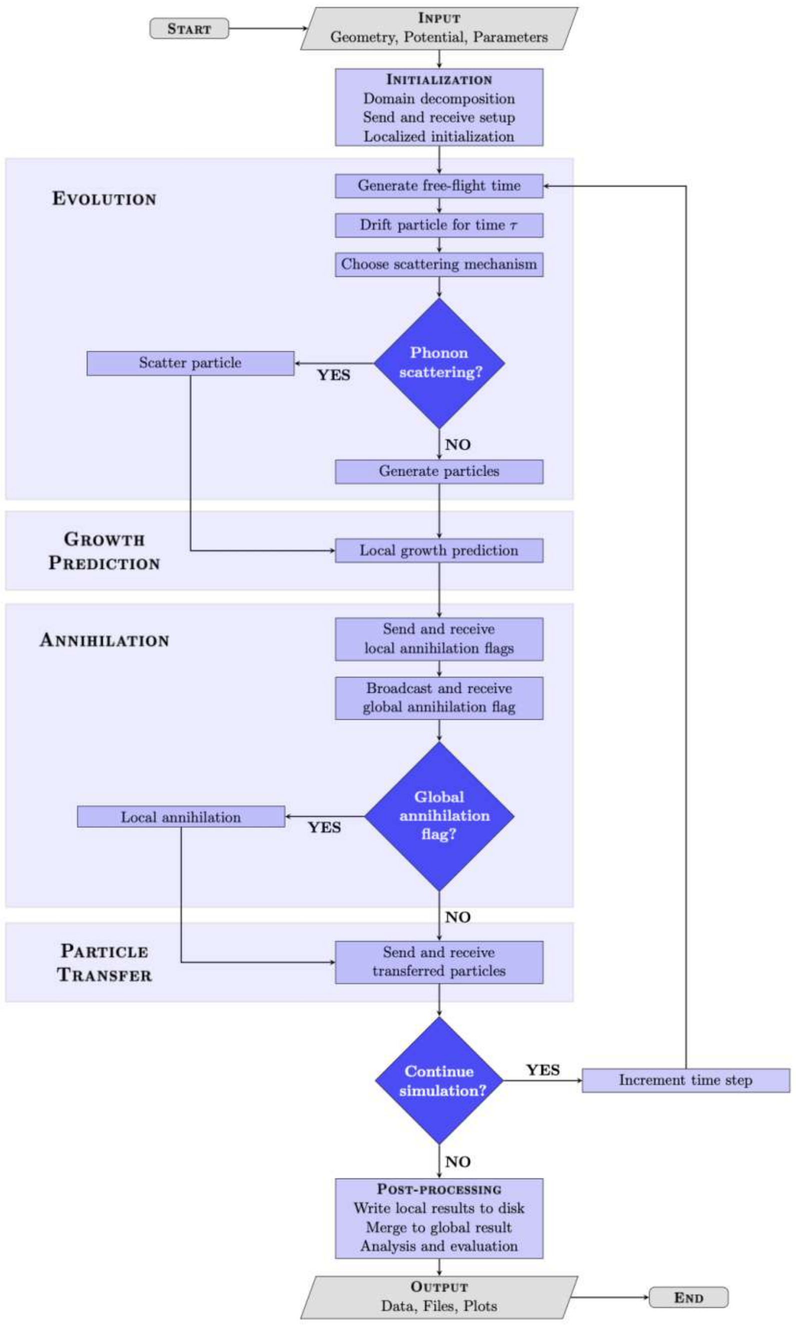


Figure 1: Flowchart of the particle Wigner simulator ViennaWD [11].

1.3 Terminology

This section provides an overview of often used terms in this paper.

A CPU (central processing unit) is an integrated circuit that executes instructions on a computer. One CPU may be made up of multiple pieces of silicone but is one integrated physical piece of hardware.

A CPU core is a single unit within the CPU which is able to execute instructions. Nowadays, multi-core CPUs are the standard.

A thread is a single virtual set of instructions given to a single processor core. There may be multiple threads executed on a single core in what is called simultaneous multithreading.

A (compute) node is the physical system including at least one CPU, memory and other potential accelerators like a graphics processing unit (GPU).

A cluster is a collection of nodes which are interconnected by a network which allows them to communicate with each other and to execute a single, parallelized program.

A mesh is a way to subdivide a simulation domain into discrete elements of a certain shape such as hexahedrons or tetrahedrons in three dimensions, necessary for most computational methods to, e.g., solving partial differential equations.

The open access software library p4est is an open source forest-of-octrees mesh management library developed by C. Burstedde, L. Wilcox, and O. Ghattas, further explored in section 2.1.

A quadrant refers to a cell in the mesh which is generated by p4est. A leaf quadrant is one which has no children. Further explanations in section 2.1.

Local in the context of this thesis refers to the part of the domain that has been assigned to a single core and all elements within that domain be it particles or quadrants. All local elements are contained in the memory of that core.

The big O notation is a way to show how the complexity of an algorithm increases as an input parameter is changed in size. $O(n)$ for example means the complexity behaves linearly, while $O(n^2)$ signifies quadratic complexity increases.

A MPI process is a single instance of a program running with MPI. Each MPI process is assigned to a core when program execution begins. These will be referred to as just processes for the rest of this work, however p4est refers to them as processor within the code comments which is left unchanged.

A rank refers to an message passing interface (MPI, section 2.2) rank that a process has within one execution of a program. No two processes have the same rank, but the specific rank number that any process has is arbitrary.

1.4 Objective

The objective of this thesis is to design, develop and benchmark a minimally viable yet representative benchmark library. The purpose of this library is to explore the suitability of using a p4est data structure for mesh management for a particle Wigner simulator such as ViennaWD. P4est is already used in a number of large and highly scalable simulations such as [12] [13] and its mesh structure is in principle compatible with ViennaWD. However, a rigorous evaluation is missing, which, however, is vital to assess the suitability for using p4est for particle Wigner simulators. In the exploration of this suitability, a number of different options for data structures and supporting methods are explored. An analysis of the performance is then conducted to find potential limitations.

2 Background

This chapter gives some background information relevant to the understanding of the open source software needed for the benchmark library as well as any supporting infrastructure which is needed for actually conducting the benchmarks. It is meant to give the reader an introduction into topics which are relevant for the rest of this thesis. The p4est library is introduced along with an overview of the MPI standard and the Vienna Scientific Cluster.

2.1 The p4est Library

The meshing software library p4est that enables the dynamic management of a collection of adaptive octrees, called a forest of octrees, on large and highly parallel systems. The term octree refers to a recursive structure in three dimensions in which each element, or octant, of the tree either has eight children, which is where the ”oct” part of the label originates, or is a leaf element. A similar structure is the quadtree, which is a 2D tree in which each element (in this case called quadrant) has either four children or is a leaf element [14]. These quadtrees or octrees are only able to represent a quadratic or cubic domain, as the root element of each tree is a cube/rectangle which is then subdivided into smaller partitions. By using a collection of trees, which is why it is referred to as a forest, with connections through faces, edges or corners this structure is able to represent a wide array of different physical domains. Each leaf element in this tree represents one element of the mesh of the domain. For simplicity, each element in this tree will be referred to as a quadrant in the rest of this thesis independent of it being a quadrant or an octant. P4est will also be referred to by its library name p4est in this thesis, however, in code snippets the functions will often be referencing p8est instead. This is a library internal naming convention where all 2D functions have the prefix *p4est_* while the 3D versions have the prefix *p8est_*. This differentiation is not entirely consistent as most datatypes and functions in 3D have the p8est prefix but not all, some are shared between the 2D and 3D versions and therefore retain the p4est prefix.

The p in p4est means parallel, meaning the library is highly scalable on large systems with a parallel processing. The overall structure of the domain and its division into trees is shared across processes while the subdivision of each octree and the quadrant data is dynamically distributed. As can be seen in figure 2 the overall domain decomposition into trees is then split by processes somewhere within each tree or between trees. Due to the way trees are generated, splitting one tree between two processes at some point in the tree means that the quadrants in each process will be neighbours of each other, so the domain is kept as localised as possible. It can also be seen that this does not work as well across trees, where the

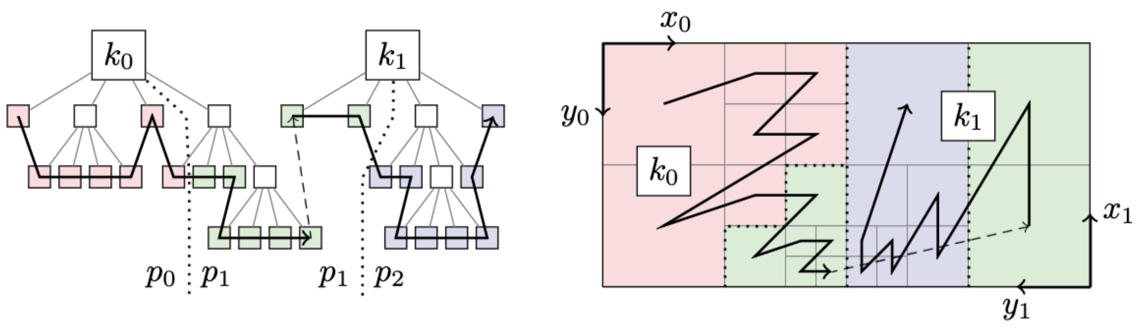


Figure 2: Schema of the p4est datastructure with two quadtrees ($k_{0,1}$) spread across three processes (p_{0-2}) indicated by different colors [14]. Arrows indicate the numbering sequence of quadrants, shown in the trees and space discretization.

orientation of the coordinate systems within trees can sometimes mean the domain of all quadrants in a process can be split with gaps. This can be rectified with certain functions provided by p4est. The parallelisation of p4est works by allowing each process to access the structure of the whole domain and identify which process is responsible for which subdomain, but the relevant mesh and simulation data is stored only locally.

The motivation for creating this library is the concept of adaptive mesh refinement (AMR) and coarsening. Adaptive mesh refinement refers to the possibility of locally varying mesh sizes, dynamically adapted within the runtime of the simulation. A finer mesh allows for the resolution of fine scale features at the cost of high computational complexity, while a coarse mesh may be numerically inaccurate [14]. Adaptive mesh refinement allows for the fine scale features to be accurately resolved by a fine mesh while allowing the rest of the mesh to be at a coarser scale, saving computational resources.

In particle Wigner simulations, the transport equation moves the particles across the domain and as these particles move the part of the domain which would benefit from having a finer mesh also moves. Therefore, a dynamic adaptation of the mesh is necessary, which adapts the mesh according to the solution process. As the particle Wigner simulation also requires particle-cell and cell-particle relationships within its calculations, it may be advantageous to have very large quadrants which contain a lot of particles, such as for annihilation events. Alternatively, it may be an advantage to have very few particles per quadrant such that load balancing can be achieved more easily, as if very few quadrants contain a large number of particles it may not be possible to achieve an optimal load balance. Both of these contradictory situations should be possible in principle, to retain a high degree of flexibility with the parallel solution process. In this benchmark both non-load balanced and load balanced states are tested, which ultimately will show why load balancing is absolutely necessary, as well as the performance when including very fine or very coarse meshes.

The structure of an octree lends itself very well to this adaptive mesh refinement as each leaf quadrant of the domain can be subdivided into eight quadrants, increasing the resolution of this part of the domain. Conversely, a node with eight leaf quadrant children can be converted into a single leaf quadrant in order to coarsen the mesh.

An additional benefit of AMR, which is also integral to the functioning of the p4est library, is the possibility to do dynamic load balancing in line with AMR. A load imbalance occurs when synchronisation points are reached by some processes earlier than others, leading to at least one process idling while others still do work, under utilizing the total resources [15]. This load imbalance may occur due to a concentration of quadrants, particles, or a combination of both in a certain process or number of processes. Adaptively refining the mesh such that each quadrant contains a number of particles within a certain range, then evenly distributing quadrants across processes, should generally lead to a well-balanced load across processes. This idea and the actual results will be further discussed in the performance analysis later in this thesis.

2.2 Parallelisation with MPI

The MPI is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. MPI enables parallel execution using the concept of message passing, also refered to as communication, between processes which allows them to share data with each other.

MPI is the de facto standard in high performance computing for usage on very large distributed memory systems, sometimes used in conjunction with other libraries which allow for shared memory parallelisation. MPI utilizes a distributed memory system, which means each processor is connected to exclusive local memory [15] and is connected to others by a general interconnection network [16]. A processor in this case is is a CPU, however typically each MPI process does not have access to the local memory of other MPI processes, even if they reside on the same physical processor. The alternative is to distributed memory is shared memory parallelisation, which means all processes have access to global physical memory through which they share variables and data structures [16]. The combination of these two, where for example the memory within a node is shared but not across nodes, is called a hybrid system. Shared memory systems are easier to develop for, however they suffer from increased latency when accessing memory and limited scalability compared to distributed memory systems [16]. The advantage of MPI is that it allows for much better scaling when increasing the number of processes due to it being based on distributed memory parallelisation, scaling up to hundreds of thousands of CPUs at the cost of increased code complexity.

Message passing means that one process sends a message which contains a certain part of its memory to another process, which then has to receive a message from this specific process. Message passing can be done using point-to-point or collective communication between processes, in a synchronous or asynchronous way depending on the need of the user. Communication can be done in a blocking or non-blocking way, blocking meaning that a process executes the communication call and stops further execution until the message is sent or received while non-blocking does not impose this execution stop. Non-blocking communication allows the user to continue execution within the program while waiting for the message to be received, which can increase efficiency [17]. Both blocking and non-blocking communication is used in the developed benchmark library.

MPI is just a standard and there are a number of different implementations of this standard, there is also a choice to be made for which exact implementation to choose for this benchmarking. OpenMPI is one of the most common libraries for this, along with IntelMPI, MPICH and MVAPICH2. OpenMPI was chosen for this implementation although there should not be any large differences in performance between MPI implementations [18] [19] [20].

2.3 Vienna Scientific Cluster

The benchmarking of this library and all performance testing was performed on the Vienna Scientific Cluster, specifically on the VSC-5 supercomputer. The Vienna Scientific Cluster (VSC) is a collection of supercomputers that support research activities in areas like physics, chemistry, meteorology, life sciences and others for several Austrian universities. The currently active systems in the VSC are MACH-2, a shared memory system, LEONARDO as part of a multinational initiative, VSC-4 and VSC-5.

VSC-4 is a system utilizing 790 water cooled nodes each with two Intel Skylake Platinum 8174 processors with 24 cores, totaling up to 37,920 cores, interconnected with 100 Gbit/s Intel OmniPath. Each node has 96 GB of memory by default, with 78 ”fat nodes” equipped with 384 GB of memory and 12 ”very fat nodes” with 768 GB. It is ranked 319 in the Top500 list [21].

VSC-5 features 710 compute nodes with two AMD EPYC Milan 7713 64 core CPUs each, totaling up to 98,560 cores for the whole system. These nodes are connected by a Mellanox HDR inifiband communication system which achieves 200Gb/s bandwidth. Each node is equipped with 512 GB of memory, with 120 large memory nodes equipped with 1024 GB and 20 extra large memory nodes with 2048 GB. It is ranked 416 in the Top500 list [22].

The VSC-5 system was chosen for the performance benchmarking of this library due in part to its higher communication bandwidth, its higher total core count and its larger memory per node. The larger memory did not turn out to be relevant in the benchmarking.

3 Implementation

This section gives an overview of the implementation of the benchmarking library and supporting implementation-specific information needed given the goal of creating a benchmark library which ascertains the performance of p4est for a particle Wigner simulator. First, an introduction to the library is given which shows how the previously mentioned building blocks, mainly p4est and the particle Wigner simulation, interact with each other to build a benchmarking process. Then follows an introduction to how p4est can be used in a particle Wigner simulation, how it handles different mesh types and which of its functions are needed and how they work. Two different approaches to storing particle data as well as an exploration of mesh data storage in p4est follow. To conclude the Implementation chapter, different strategies for particle generation, deletion, particle-cell/cell-particle relationships and communication are laid out before all of these topics are collected and the specific steps in the benchmark are explained.

3.1 The Benchmark Library

The goal of this library is to create a minimal working example which encompasses all functions that a Monte Carlo particle Wigner simulation would require to run without implementing the actual simulation in order to ascertain the performance limitations of using p4est as a mesh management library for this approach. As such, a number of steps of the ViennaWD simulator laid out in figure 1 are not implemented, such as the photon scattering or growth prediction steps, but for those relevant to the management of particles and the mesh a minimal implementation is created which encompasses all steps needed to set up the data for the simulation. This means that if a particle Wigner simulation were to be implemented, while this library does not provide a framework for such an implementation the implementation of the library can be useful as a stepping stone which includes all concepts needed in the simulation but would need significant modification to apply those concepts to a complete Monte Carlo particle Wigner simulation.

In order to give an overview of the benchmark, a schematic showing the interaction between the different concepts previously introduced and the particle Wigner simulation is shown in figure 3. The benchmark library makes use of p4est and its mesh management but does not include the particle Wigner simulation parts which govern the interaction between the field data and the particles. The benchmark library still uses p4est and the mesh in its operation but does not include the field data as its only used within the particle Wigner simulation. The two arrows indicating the impact of field data on particle data and vice versa are part of the particle Wigner simulation and are represented in the benchmark by providing a way to access the quadrant in which a particle resides and the particles which are contained within a quadrant. The linking of these elements (quadrants and particles) as particles move through the domain is relevant to performance as a search algorithm that connects these elements slows down as the number of particles and quadrants increases, so the rate at which this slows down must be explored within the benchmark.

As the benchmark runs in a distributed memory environment, MPI is used to distribute all data across the different processes both for the mesh and for all particles. The way that the mesh is distributed across processes is determined by the benchmark library through functions provided by p4est, but these functions only give a general target (balancing load for example) the details of which is handled by p4est. The distribution of particles across processes is managed by the benchmark library and the options for this are explored in section 3.8. The interactions between the particles and the field data is not simulated like in the particle Wigner simulation in the benchmark library, however these interactions require the particles and field data to be connected in some way. The connection between these two data structures is vitally important to the simulation so it is included in the benchmark library. The connections between particles and the field data are referred to as particle-cell and cell-particle relationships and the way these connections are formed is explored in section 3.7 while the storage of these relationships is explored in section 3.4. The benchmark library as provided consists of a number of functions that each accomplish a certain task in support of the particle Wigner simulation, which are laid out in section 4. Importantly, the library is not optimized for a certain simulation setup or a given particle distribution, so a number of different possible strategies are laid out in each part of chapter 3 and then benchmarked against each other in chapter 4.

3.2 Meshes in p4est

Meshes in p4est are handled as a distributed forest of octrees. Each of these trees describes a cubic domain, which is then subdivided into quadrants [5] which make up the mesh. In order to connect these trees, a connectivity structure is established which determines how each tree is located in space with reference to the others as well as how they are connected along faces, edges and corners. P4est creates these forest and connectivity structures once when the simulation is started but does not allow for them to be modified once they have been established, only the actual trees may be modified. It does this by loading either a mesh from a file, using Abaqus .inp file format for hexahedral and tetgen .node/.ele format for tetrahedral meshes or by using an internally provided sample geometry.

The sample geometries provided by p4est that can be loaded without any input file are for example a cube, torus or pyramid among others. These simple geometries are meshed using a hexahedral mesh with the least amount of elements possible, for a cube geometry only a single tree is created which contains a single quadrant on its creation. If one does not wish to use a more custom domain, these are the fastest and easiest way to obtain a working geometry.

The alternative is to load a mesh from an input file. This allows the user to load any complex geometry that they may desire beyond simple domains, which is one of the motivations for this thesis as

[5] Technically speaking, these should be called octants in the here considered 3D case. Regardless, quadrants is used which is in line with the software API.

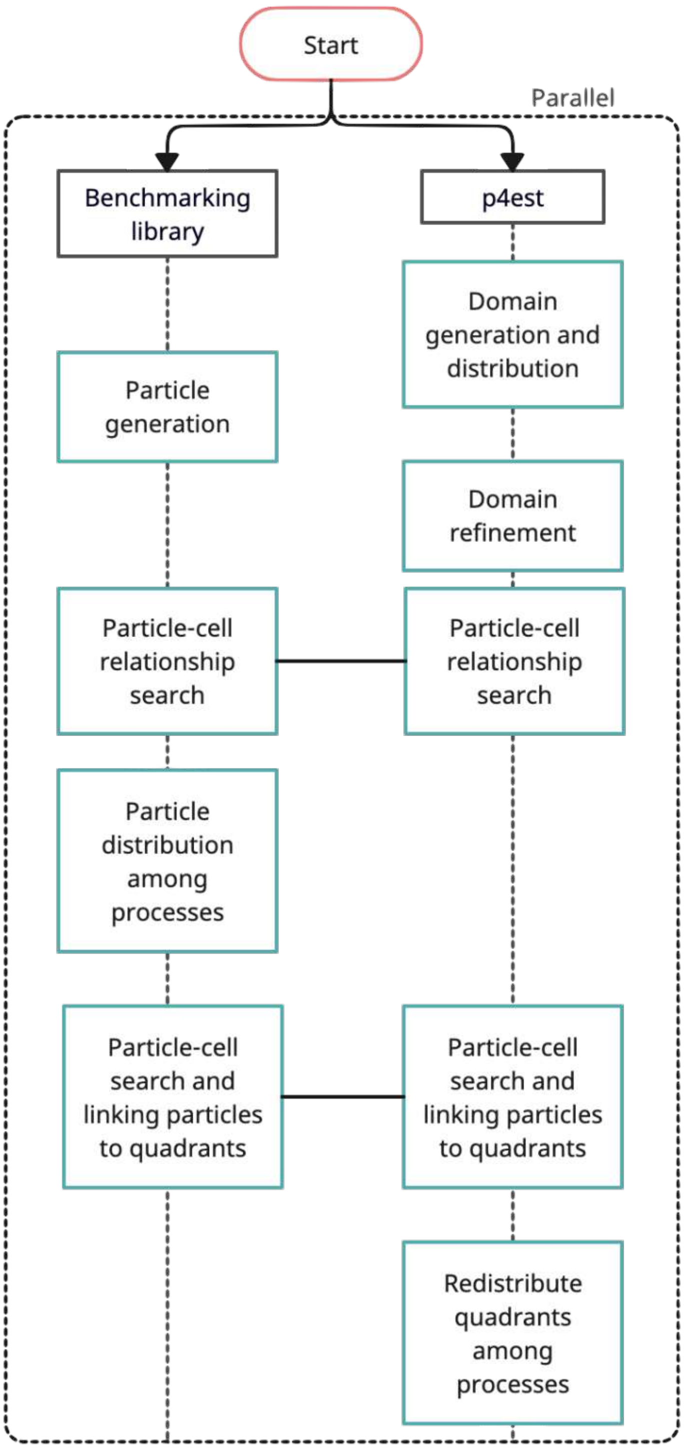

Start
Parallel
Benchmarking library
p4est
Domain generation and distribution
Particle generation
Domain refinement
Particle-cell relationship search
Particle-cell relationship search
Particle distribution among processes
Particle-cell search and linking particles to quadrants
Particle-cell search and linking particles to quadrants
Redistribute quadrants among processes

Figure 3: Schematic showing the interaction of the benchmark library and p4est.

it is a desirable feature in the next iteration of ViennaWD. Therefore, the ways in which a loaded mesh influences the performance of the benchmark library were explored to find limitations which may cause issues.

The first limitation which was found to impact the performance of the benchmark was when loading highly complex meshes. When loading a mesh which contains a large number of quadrants, even when the domain is very simple such as a cube, loading this mesh and generating a connectivity structure is a very slow ordeal. Creating a cube mesh which contains 8^6 quadrants by generating such a mesh in a typical mesh generator for example the Abaqus software and then loading it into p4est is several orders of magnitude slower than simply creating a cube mesh containing a single quadrant and then splitting each quadrant into eight quadrants six times which also creates 8^6 quadrants. The second issue with loading highly refined meshes is that when loaded into p4est, each cell within the mesh is treated as a different tree in the forest of octrees. This has a number of effects, primarily that p4est is unable to merge multiple trees into one larger tree such that the mesh cells which were loaded into the datastructure be merged into a larger cell. This means that the number of cells in the mesh which is loaded provides a floor for the number of quadrants which exist in the p4est mesh structure. Additionally, the connectivity between each of these trees needs to be stored within p4est which consumes memory for no real reason and may slow down certain operations when for example the neighbours of a quadrant need to be determined.

This means that if one would like to load their own custom domain into p4est this domain should be subdivided as coarsely as possible. If there are certain areas of interest which require a certain shape of mesh around them for example, of course these structures should be represented however they need to be to meet the specific requirements within the mesh. But if there is no specific requirement to the mesh in its domain or shape, the discretization should be as coarse as possible.

The loading of two different types of mesh, the hexahedral and tetrahedral mesh, are described in the following sections. A hexahedral mesh is preferred for the particle Wigner simulation but if one may have a desire to load a tetrahedral mesh, as this is possible in p4est, this option is also explored.

3.2.1 Hexahedron

Hexahedral meshes can be loaded using the Abaqus .inp file format. This format stores all nodes as numbered elements with their coordinates and then the connections between nodes as lines, surfaces and volumes which are all numbered and determined by which nodes belong to them, two for lines, four for surfaces and eight for volumes. A volume is the term Abaqus uses for a cell or quadrant in the mesh. Each of these elements is of a certain type, which determines how they are shaped and for Abaqus use, how their interpolation points are set. For p4est however, only the numbered nodes with their coordinates as well as the surfaces for 2D or volumes for 3D are relevant.

There are a number of different element types that can be used for loading of 2D meshes, namely the C2D4, CPS4 and S4 element types. These elements are named in their Abaqus naming convention, which means a C2D4 element is a continuous, 2 dimensional, 4 node element which is described by having all 4 nodes on the corners of the element. A CPS4 element is a continous, plane stress, 4 node element which means that the 4 nodes are not placed at the corners of the element, but instead at the integration points of the element. A S4 element is a shell element described by its 4 nodal corners.

For a 3D mesh, there is only one element type available for loading which is the C3D8 element. This element is defined as a continuous element in 3 dimensions which is described in the file structure by its 8 corner nodes. This type of element is ideal for a p4est data structure as an eight corner element is easy to transform into the quadrants that p4est stores. As each of these volume elements may be of any shape, they are treated as their own tree in the p4est datastructure. A simple cube mesh using these types of elements is shown in figure 4, which was automatically split across 64 processes using p4est.

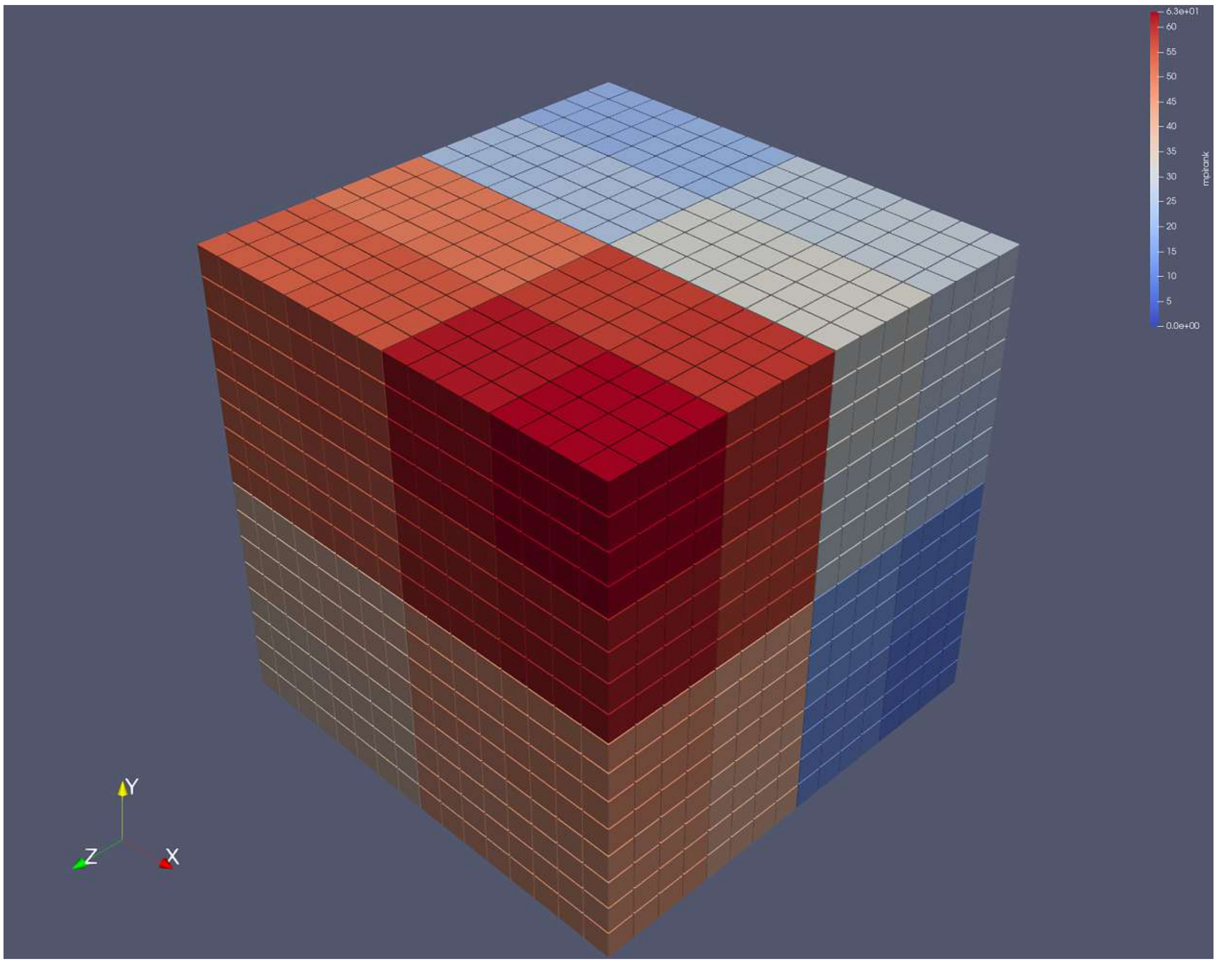


Figure 4: Hexahedral mesh colored by rank, 64 processes.

3.2.2 Tetrahedron

While tetrahedral meshes are not optimal for solving the fast fourier transformations which help the solving of the particle Wigner simulation, it is still possible and loading a tetrahedral mesh would be very advantageous in order to be able to more accurately map non-regular domains. P4est does provide functionality for loading tetrahedral meshes based on a different file structure than hexahedral ones, using .ele and .node files generated by the *tetgen* software. These two files describe the locations of the nodes in the mesh in space, and how these nodes are connected to each other, but there are no different element types possible here unlike the Abaqus input files.

In conducting tests for the loading of tetrahedral meshes it was found that the functionality is more experimental than functional and that are a number of issues with the loaded tetrahedral meshes that preclude this functionality from being useful. First and foremost, the way that p4est numbers elements within the mesh only works for hexahedral meshes. P4est numbers quadrants according to a z-numbering scheme which can be seen in figure 2, the exact design of this is not necessary for the explanation in this thesis just the fact that it breaks down for tetrahedral meshes. This leads to a situation where the elements are effectively randomly distributed among the processes as can be seen in figure 5, which defeats the purpose of any locality that exists in the system. To make matters worse, when attempting to refine the tetrahedral elements into sub-elements, hexahedral elements are created out of these tetrahedrons as in figure 6, which fixes the locality and numbering problem but would require the software to be able to distinguish between tetrahedral and hexahedral elements within one simulation which does not make sense. Furthermore, the built in algorithms for balancing the tree and ensuring compatibility across faces and edges do not work with this messed up numbering, which means they are all useless. It is possible that in further updates of the p4est library, a tetrahedral mesh will be usable with full functionality for this tetrahedral mesh, but at the moment this is not possible. If and when this functionality is complete the performance characteristics of this new version of p4est should be explored.

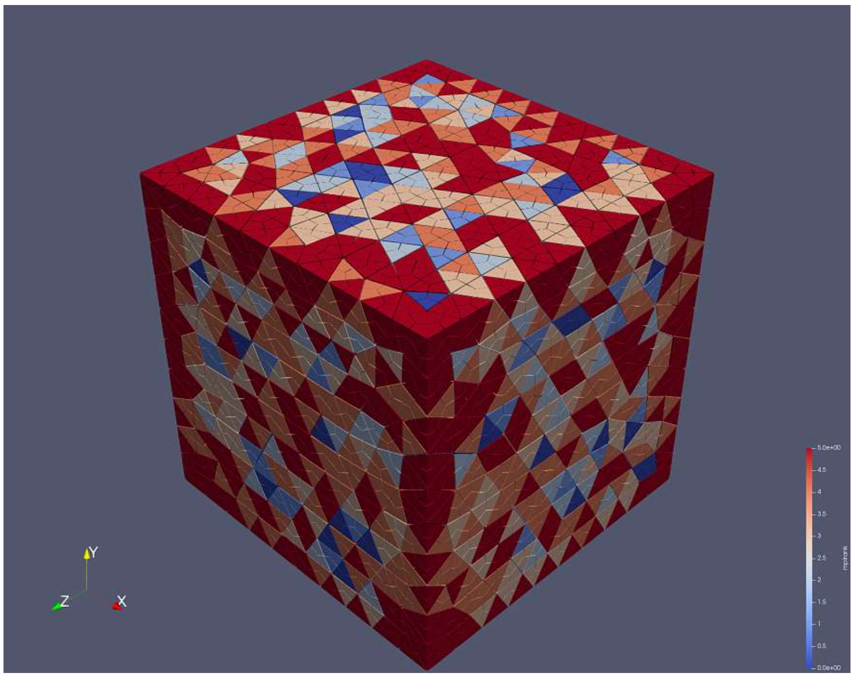


Figure 5: Distribution of tetrahedral quadrants across five processes colored by their MPI rank.

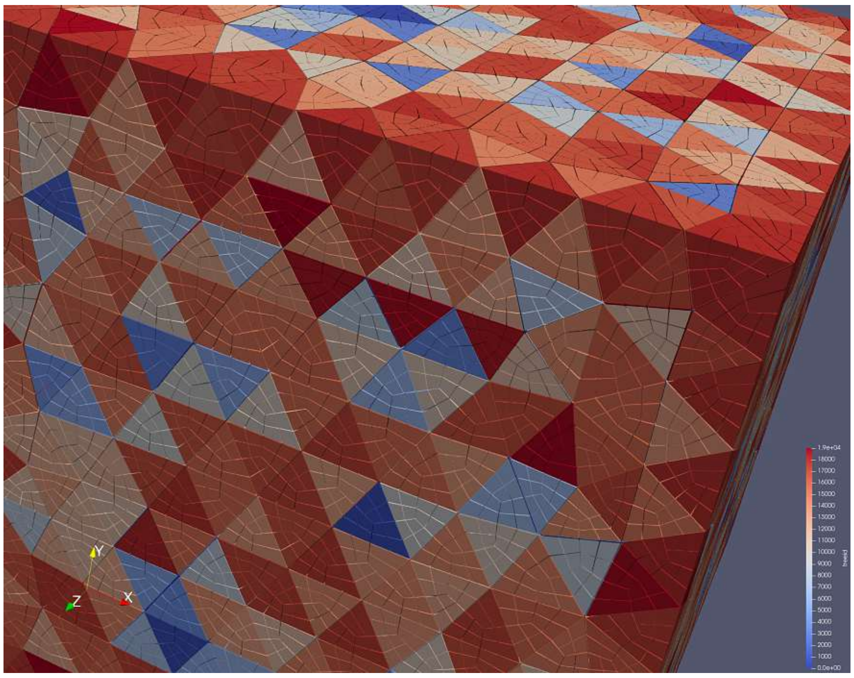


Figure 6: A tetrahedral mesh where each element has been refined once. Parts of the mesh in different colors are parts of different trees in p4est

3.3 Relevant Functions in p4est

Once a mesh has been loaded either from a file or from the sample geometries p4est has provided, and a connectivity between trees has been established, the generated mesh may be used within the execution. A core part of the functionality of the benchmark library are functions provided by p4est, as these are necessary for managing the mesh and domain decomposition among processes. Not all functions of the p4est library are used here but a number of them are presented to explain their functionality and how it relates to a particle Wigner simulation, specifically the $p4est\_search\_all$, $p4est\_refine$ and $p4est\_partition$ functions.

Listing 3.1: The $p4est\_refine$ function

```
/*param [in,out] p8est The forest is changed in place.
param [in] refine_recursive Boolean to decide on recursive refinement.
param [in] refine_fn Callback function that must return true if a quadrant shall be
    refined. If refine_recursive is true, refine_fn is called for every existing and
    newly created quadrant. Otherwise, it is called for every existing quadrant.
param [in] init_fn   Callback function to initialize the user_data of newly created
    quadrants, which is already allocated.*/
void                 p8est_refine (p8est_t * p8est,
                                   int refine_recursive,
                                   p8est_refine_t refine_fn,
                                   p8est_init_t init_fn);
```

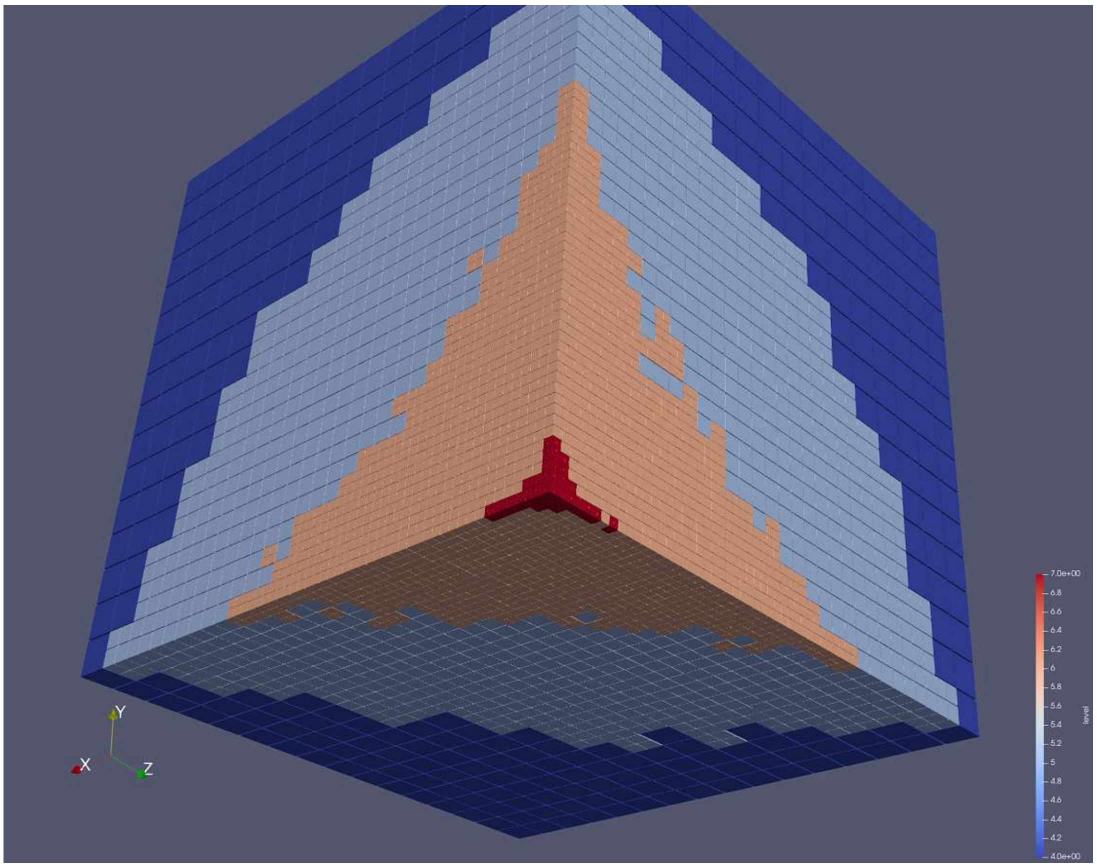


Figure 7: Mesh which has been refined to match a certain particle distribution. Colors indicate the refinement level of the quadrant which is a representation of its size.

The *p4est_refine* [6] function shown in snippet 3.3 provides a functionality that allows the user to refine the currently existing mesh based on certain user-defined criteria per quadrant. Refine here means splitting a leaf quadrant into eight (or four in the 2D case) new leaf quadrants. It does this by iterating through all leaf quadrants in the domain and passing them as an object to this function, which is then executed. This allows for the user to make a decision based on certain data associated with this quadrant, for example the number of particles contained within it or its position. This is core to the adaptive mesh refinement functionality, as this is what allows the mesh to be finer for certain parts of the domain than others. There is also a coarsening function in the library called *p4est_coarsen* which does the opposite of refining by taking eight (or four in the 2D case) leaf quadrants and letting the user make a decision to either combine them into their parent quadrant or not.

The refine and coarsen functions primary use within the particle Wigner simulation is to provide a finer resolution of the mesh where it is required in the simulation. In a situation where the particle distribution is non-uniform as in figure 10, one may refine the mesh like shown in figure 7.

The *p4est_refine* and *p4est_coarsen* functions are essential to the mesh refinement matching up with the areas of interest defined by the user or the physics in the particle Wginer simulation, however as

[6]This function is called *p8est_refine* in the code snippet, but these are equivalent functions acting on a 2D and 3D geometry respectively.

there is no simulation in the benchmark these functions are used in a primitive way simply refining every quadrant in the domain when they are called. The implementation of more sophisticated refine and coarsen functions depends on the implementation of the probability density function and other implementation specific information. This should not be an issue in the performance benchmarking however, as the performance is related to the total number of quadrants either locally or globally, not their size, and the refinement itself is a quadrant-level function, meaning it is called once for each quadrant, and the performance of quadrant-level functions is explored based on other, similar functions later on.

Listing 3.2: The *p8est_search_all* function

```
void p8est_search_all (p8est_t * p4est, int call_post,
p8est_search_all_t quadrant_fn, p8est_search_all_t point_fn,
sc_array_t * points);
```

Listing 3.3: The *p8est_search_all_t* function template required for the *p8est_search_allfunction*

```
typedef int (*p8est_search_all_t)
(p8est_t * p8est,            //the overall p8est object
p4est_topidx_t which_tree,  //an integer to
p8est_quadrant_t * quadrant,//the quadrant in question
int pfirst, int plast,      //two integers which are the MPI rank of the lowest and
    highest rank MPI process which own a part of this quadrant
p4est_locidx_t local_num,   //negative if not a leaf, otherwise the index of this
    quadrant in storage array
void *point);               //pointer to the particle in question
```

Snippet 3.3 shows the *p8est_search_all* function as it is defined by p4est, and snippet 3.3 shows the function template which is required to be passed to the *p8est_search_all* function. The particle and quadrant search functions must follow this template, as these functions are called within the *p8est_search_all* function.

The *p4est_search_all* function is the most important p4est function used in this benchmark library. It allows the user to connect all quadrants to all local particles by iterating through all quadrants and all local particles, and then calling two user defined functions. The first user defined function will be referred to as the quadrant function as it is called every time there is a new quadrant that needs to be iterated through, while the other is called for each new particle and therefore referred to as the particle function. These two functions then allow the user to do the required calculations or setting of flags, and either stop the iteration for this specific quadrant/particle or continue it. The specific way in which these calls are ordered is not accessible and strictly defined in the p4est library, but there is an alternative *p4est_search_local* function which only takes into account the quadrants which are local to the processor instead of the entire domain. This function is what allows for the cell-particle and particle-cell relationships to be determined, specifically it is used once to determine for each particle which processor contains the domain it is located in to allow for communication to happen, as well as once each processor has all particles within its domain the creation of a list of particles contained within each leaf quadrant.

This kind of functionality, iterating through two lists of objects, either with *p4est_search_all* or *p4est_search_local*, and comparing them to one another gets expensive in runtime as the size of the lists increases, as it is of $O(n \cdot m)$ complexity where n and m are the number of particles and quadrants respectively.

The *p4est_partition* function is responsible for the redistribution of the mesh across the domain. It does not change the mesh in any way, it uses a weighting function which operates on each quadrant to return a weight for each quadrant. P4est then uses these weights per quadrant to redistribute all quadrants across all processes. The resulting domain decomposition still ensures that a processor has quadrants which neighbour each other to preserve locality.

3.4 Mesh Data Storage

A very important part of any particle Wigner simulation is the usage of quadrant level data. This data includes field variables such as the potential energy or material properties and whatever variables would be needed additionally in the implementation itself. For the benchmarking of the framework, the important data stored on each quadrant is a list of all particles contained within this quadrant.

The quadrant data object within p4est contains the coordinates of the "bottom left" corner of the quadrant, the refinement level and therefore size of the quadrant, a number of variables to keep track of ghost and transformed information, and three variables which are reserved for the user: a double, an int and a pointer to a custom quadrant data object. This quadrant data object can be completely customised by the user and its size is then passed to p4est during initialisation, as it is stored by p4est in an *sc_array* type of array however it cannot be changed in size later on. This quadrant data object in the benchmark contains all particles present within the quadrant and a weight.

This means that all objects which represent this quadrant level data must be of constant size, so for example a vector cannot be just stored within the quadrant data object. This is a hindrance as it would be very useful to know which particles reside within a quadrant for a local annihilation event for example. One possible solution to this problem could be to simply include a sufficiently large array within the quadrant data such that storage should always be possible. A more elegant, but also potentially more complicated way to circumvent this issue is to simply include a pointer to a vector in the quadrant data object. When moving quadrants between processes these pointers would still point to the same memory address, but on a different process which is not allowed and would cause a segmentation fault. Additionally, not destructing the vectors these pointers point to would cause memory leakage. This would typically be taken care of by destructors in modern C++, however as the p4est library is written in C no destructors are available without modifying the library. Therefore this destruction of vectors and pointers must be taken care of manually, which is implemented in the benchmark.

Similarly, no constructor is accessible for these data objects and therefore the quadrant data vectors cannot be automatically initialized. The way this problem was solved is by creating an initialisation and destruction function, which clears or initializes all quadrants on the local processor. This means that the destruction function must be called before any modification or redistribution of the domain is done, as otherwise there may be memory leaks or segmentation faults. This is not very difficult to do as the functions which would do this modification are mainly the *p4est_refine*, *p4est_partition* and *p4est_coarsen* functions so therefore wrapping these in a destruction and initialisation call would ensure no memory leakage.

Additionally, as there is no inheritance between p4est quadrant objects, when the mesh is refined the newly created quadrants do not contain any data. This means all field data as well as particle connection data would need to be created again to be then saved on these quadrants. An initialization function can be called when refining the mesh which allows the user to automatically generate data on this quadrant, however there is no reference to any data of this quadrants parent within the initialization function so no inheritance is possible at the moment. An additional feature that allows this to happen would be very convenient to the overall functioning of the particle Wigner simulation, but is not part of this benchmark library and therefore mentioned only to give context.

There is a minor inconvenience here as the functions mentioned rely on the weight of a quadrant for their logic. The weight is a measure which can be defined by the user and represents how much computational complexity a quadrant contributes to the overall simulation. This weight is required for either refining or coarsening of the mesh, triggering either when the weight goes above or below a certain threshold, and distributing the mesh across processes. In the benchmark, the weight is fully dependent on the number of particles within the quadrant for load balancing purposes, however it can include other contributing factors in a particle Wigner simulation. The vector storing the particle indices, which in the benchmark defines the weight, gets destructed before the functions are called. There is another workaround to this by simply saving the size of the vector as a dedicated weight variable, which would then not be deleted as it is always present in the quadrant data. This weight saving step can be included in the destruction step, however it is not done in such a way for this library as the weight saving step may or may not be necessary every time a deletion takes place. Additionally, these steps only operate on processor local quadrants which are typically not very large in quantity and therefore operate quite fast, which will be further discussed in section 4.9.

Listing 3.4: The *quadrant_data* object

```
typedef struct quadrant_data{
  std::vector<int> * contained_particles;
  int weight;
}
quadrant_data_t;
```

Snippet 3.4 shows the quadrant data stored on each quadrant. Field data is not needed for benchmarking and therefore not included, but would be stored within this object.

3.5 The Data Structure

The overall data structure of the developed benchmark implementation relies on a single object, the *particles_global* object, to encompass all data in the benchmark. The structure of this object is shown in snippet 3.5. The reason why there is one object encompassing everything is that p4est allows the user to store a single pointer within the p4est data structure, which is then able to be accessed from within any p4est function. This is necessary as there are a number of functions called by p4est, such as *p4est_search_all* and the refine functions which do not allow the user to pass any arguments to it, so any user accessible data needs to be accessed through this one pointer within the p4est object as this is available within those functions.

It should be noted that as this is a distributed memory system, the "global" reference in the name of the object does not mean it is shared across processes. Each process has a separate instance of both the *particles_global* as well as *p4est* objects.

Within the single *particles_global* object, there are two pointers which reference back to p4est, specifically the overall *p4est* object as well as the *p4est_connectivity* object. This referencing loop allows all data to be accessible from anywhere within the program where either the *particles_global* or the *p4est* object are available. Ensuring that this referencing loop is correctly setup is done by only allowing the *particles_global* object to be created by a constructor which also creates the *p4est* object and passes a reference to itself to it, or takes an existing *p4est* object and stores a reference to itself in that pointer.

Besides the *p4est* and *p4est_connectivity* objects, the *particles_global* object contains a number of auxiliary variables such as the total number of particles in the benchmark, the round size and temporary storage for coordinates, which is needed for the cell-particle and particle-cell relationships, and the particle data. Particle data is stored in a vector of vectors as a double in this benchmarking implementation, but not all particle data may be floating point but instead other types such as integers. For example, is a vector of integers in the benchmark library which represent the id of a processor, used for marking a particle for deletion in this case, but this id may be used for any other data in a particle Wigner simulation. If more than one integer is required per particle, this will need to be expanded to a vector of vectors as well. These two vectors of vectors are going to contain a large amount of data, as will be shown in 5, therefore their storage is given more consideration.

Listing 3.5: The *particles_global* object

```
class particles_global
{
    int mpirank;                    //the rank of this processor
    int mpisize;                    //the total number of processors
    MPI_Comm mpicomm;               //the MPI communicato

    p8est_connectivity_t* conn;     //the p4est connectivity object
    p8est_t* p8est;                 //the p4est object

    int numparticles;   //total number of particles across processors
    int roundsize;      //roundsize if round based communication used
    int maxlevel;       //maximum refinement level in the mesh
    double lxyz[3],hxyz[3],dxyz[3];     //helper variables

    std::vector<std::vector<double>> particles_data;  //particle data
    std::vector<int> id;                          //more particle data
    std::vector<int> particles_target_processor;      //vector that
                    //stores the target processor for each particle
    std::vector<int> numsend,numrecv;   //helper variables for communication
    sc_array_t* index_array;    //the sc array needed for search functions
};
```

3.5.1 Optimizing Particle Data Storage

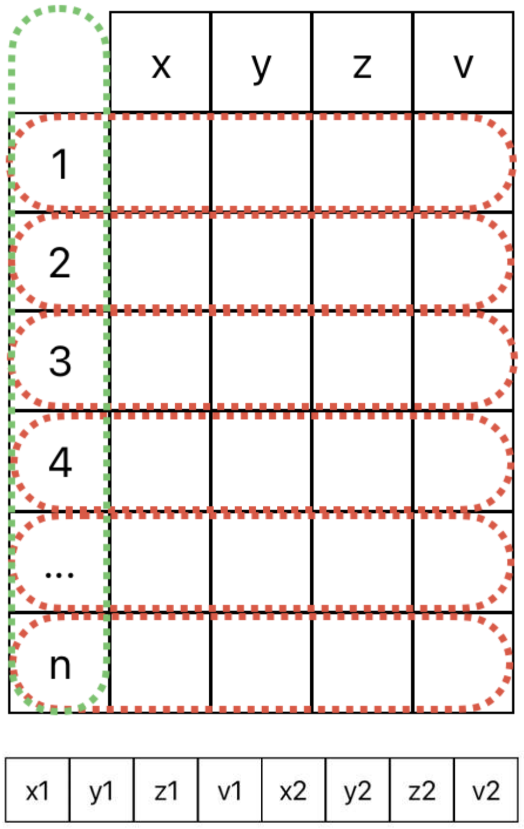
The storage of particle data requires careful consideration as the particle data is the largest part of the data structure and the one most relevant for performance. Each particle requires six double precision floating point variables to describe its three coordinates in real and phase space, as well as a separate integer to represent its generation and a boolean to represent its sign for a signed particle Wigner simulation. Additional variables may also be required, which would be easily added. All of these variables will be required once for each particle, but may be accessed in different parts of the particle Wigner simulation more or less frequently depending on the actual implementation of the simulation. All of the particle data together may be thought of as a two-dimensional array which is eight elements wide, not considering that these may be of different data types, as that is the number of variables for each particle, and as long as the number of particles on the respective process. The easiest way to store two dimensional data in C++ is a vector of vectors.

There are fundamentally two ways of storing the $8 \cdot n$ array in this vector of vectors, either creating a vector which contains eight vectors, each representing a certain variable and containing the data of all n particles, or creating a vector which contains n vectors, each of which represent a specific particle and holds each of its eight variables. These two ways of organising data are typically referred to as either as a Structure of Arrays (SoA) or Array of Structures(AoS). A SoA type memory layout means there is one structure which holds in this case eight arrays, each of length n while AoS means we have an array of length n each containing a structure. The two layouts are schematically shown in figure 8

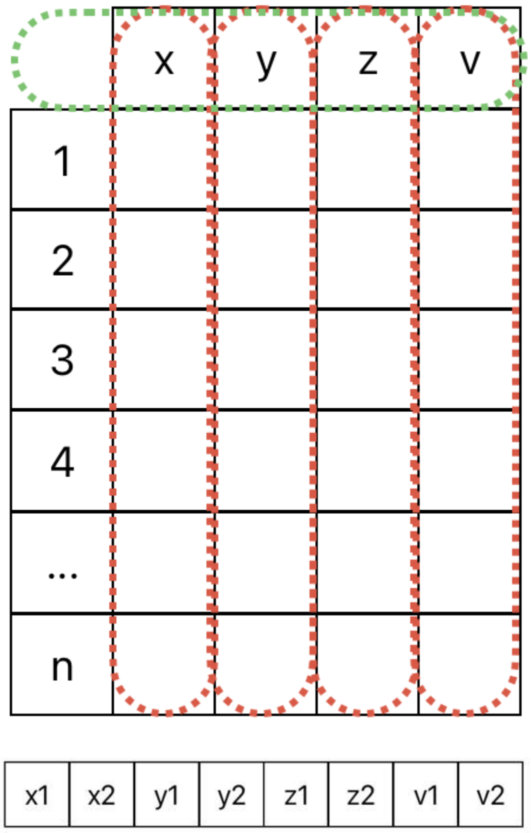
These different memory layouts are relevant to the performance of the particle Wigner simulation in the case when it is memory bound, not compute bound. Memory bound means that the access of data from memory is limiting the performance of the system as it takes longer to load the data than to process it, while compute bound means that the computation speed is limiting performance as it is slower than the memory access. The memory layout is relevant to this because memory is read in cache lines, meaning that a certain size of data (typically 64 bytes or 8 doubles) is read from the onboard memory into the cache of the processor. This process is illustrated in figure 9. If the data is stored in such a way that the relevant data for a number of computations is on the same cache line, this is much more efficient than loading a number of variables from multiple cache lines, as the number of cache lines needed to load is proportional to the time needed for loading them. Therefore, if the relevant data for a calculation is stored in an optimal way, the performance of a memory bound particle Wigner simulation can be improved. Further performance analysis in section 4.3 indicates that SoA type storage is more optimal than AoS type storage, although this is not conclusive.

As there is no simulation being run in this benchmark, the two different methods of data storage are explored to find any meaningful differences in performance and implementation which would need to be taken into account in the application of the library.

As the function *p4est_search_all* is imported from the *p4est* library, it requires the particle array through which it is iterating to be of the *sc_array* type from the sc library. This works for the AoS method of data storage, as the array can be of a type which encompasses all particle data, this is not possible with a SoA type storage as this requires a number of arrays which each need to be dynamic in their size, something which is not possible for the *sc_array* type as it requires fixed size elements. In order to ensure consistent performance benchmarks, the choice was made to create an index array of ints



(a) Array of Structures (AOS)



(b) Structure of Arrays (SOA)

Figure 8: Two potential data structures compared, red showing the inner vectors and green the outer vector. Below is the layout of data in memory.

which adjusts its size to the number of particles on the process, but only contains its index as its value. This means that this index array can be passed to the *p4est_search_all* function and the index which is available within the function is used to access the actual data structure.

The result of this is that the *particles_global* object contains one *sc_array* which is the index array and the particle data structure which is either SoA which means a number of *std* :: *vectors* of type double each representing one value of the particles data, or AoS which means it contains a single *std* :: *vector* of type *particle_data*, an object which contains all particle data.

3.6 Particle Generation and Deletion

Particle generation and deletion is a core part of the particle Wigner simulation as the annihilation steps delete a large number of particles and generation happens within each timestep as well as can be seen in figure 1. In order to provide different scenarios for the benchmarking there needs to be different distributions across the domain which test how the particle Wigner simulation would perform in an optimal and non-optimal situation. In order to achieve this, two different particle distributions are used

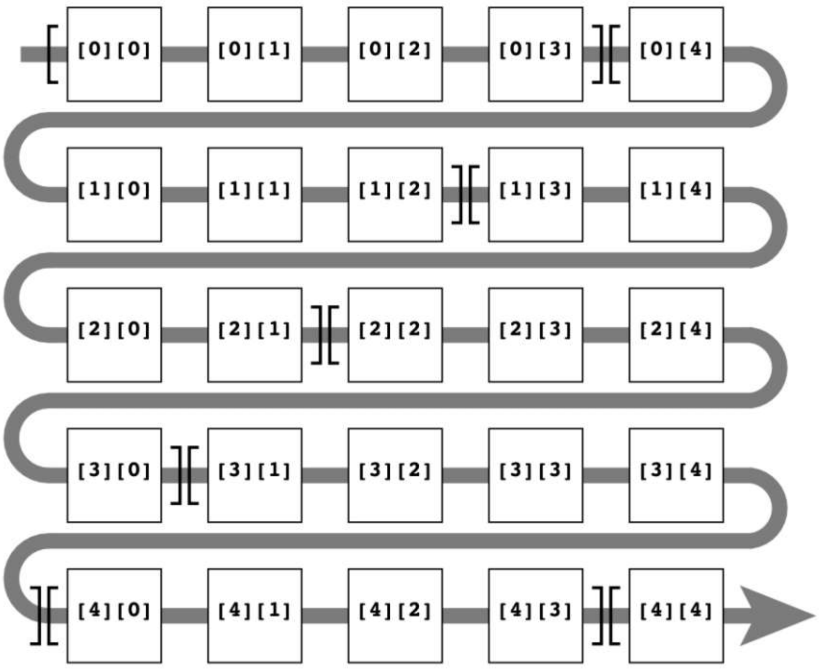


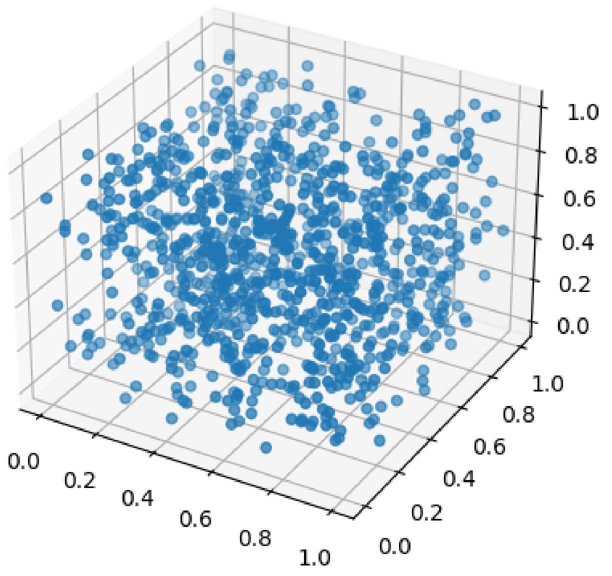
Figure 9: Data in a matrix being read in the typical fashion of the C language. The arrow indicates the order in which values are read, brackets indicate the data being read within a single cache line. [15]

for the benchmarking referred to as uniform and non-uniform for the rest of this thesis. Both of these are shown in figure 10.

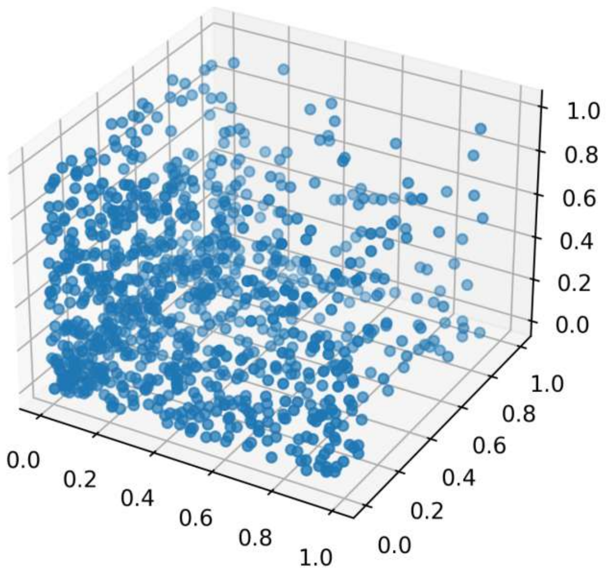
For the uniform distribution, particles are generated across the entire domain in a uniform manner. As the domain in the benchmark is a cube, each particle is generated by assigning it a value between the ”bottom left” (0,0,0) and ”top right” (1,1,1) of the domain for its coordinates and a normalized value between zero and one for each coordinate in the phase space as the actual values for the phase space are not relevant to the benchmark. In the non-uniform case, the values for the particle coordinates are also generated between these two numbers but are then squared. This produces a distribution which is very concentrated in in one corner of the domain. This concentration, with a uniform distribution of the domain across processes, means that certain processes will have many more particles within their domain than others. An example of this is that when using 64 processes with 10^7 particles in the entire domain, process 0 has $135 \cdot 10^6$ particles within its domain while process 63 has only $25 \cdot 10^3$.

There are a number of ways to generate these particles. One way to generate these particles is in what is called a master-worker model, which means that one process is chosen to generate all particles across the entire domain and then distribute them across processes. Another way is to generate particles in a distributed way which means that each process generates particles across the whole domain, and then all processes communicate particles which are not on their own domain to the processes on which domain they reside. Both of these generation strategies are laid out in section 4.1.

A third way to generate particles which is not explored in this thesis is to generate particles in a distributed way such that each process generates only particles on their own domain, which means that there is no need to communicate particles after the generation step as each particle is already on its



(a) uniform



(b) non-uniform

Figure 10: Particle distributions

correct process. Generating in such a way is very dependent on the distribution function according to which they are generated but no such function exists in the benchmark. The Monte Carlo particle Wigner simulators typically generate particles in a master-worker fashion because of the requirements such as that the signed particles are always generated in pairs with opposing signs, and the abstraction of this master-worker model to a distributed one is very straightforward so therefore was included in the benchmark. Meanwhile this distributed but local generation would have expended significant resources to develop without appreciably increasing the efficacy of the benchmark to find performance issues and test the p4est library in the Monte Carlo particle Wigner context, so therefore is only mentioned here as an option to be explored by readers who would like to apply the learnings of this thesis.

Deletion is also very important to the particle Wigner simulation as well as the working of the library. Fundamentally, there are two ways of handling deletion either by doing it instantaneously when it is determined that a particle should be deleted or by marking them for deletion and then calling a specific function which deletes all that were marked. The option of deleting all at once was chosen in this instance to be able to measure performance of this deletion step.

Deletion by removing single elements out of an array is very inefficient. When one element is deleted like this, all other elements behind the deleted one in the array have to be moved in memory. If multiple elements need to be deleted then each deletion causes this cascading moving of all elements behind it. A more efficient way of doing this, which is how the deletion function is implemented for other parts of the benchmark and shown in snippet 3.6, is to swap all elements to be deleted with the last one in the array which does not need to be deleted. This allows the array to simply be shortened and the number of elements moved is limited to twice those which need to be deleted, which is much better than the earlier mentioned alternative. The worst case scenario, when all elements need to be deleted, would produce $O(n)$ scaling of the number of movements in memory while the more simple implementation would produce $O(n!)$ scaling which is much worse.

Listing 3.6: Deleting particles using *std* :: *swap*

```
void particles_global::delete_marked(){
  int last=this->id.size()-1;
  for(int i=0;i<=last;i++){
    if(this->id[i]==-1){ //particle has been marked for deletion
      std::swap(this->id[i],this->id[last]);
      for(int j=0;j<this->particles_data.size();j++){
        std::swap(this->particles_data[j][i],this->particles_data[0][last]);} //sawp data
     for each SoA vector
      i--; last--;
    }
  }
  this->id.resize(last+1);
  for(int i=0;i<this->particles_data.size();i++){
    this->particles_data[i].resize(last+1);}
  sc_array_rewind(this->index_array,last+1);
}
```

3.7 Cell-Particle and Particle-Cell Relationships

Particle-cell and cell-particle relationships are important to the particle Wigner simulation as the physics requires an interaction of the particle data with that of the cell in which the particle resides.

As mentioned in section 3.3 the particle-cell and cell-particle relationships are determined using the *p4est_search_all* function, which allows the iteration through an array of custom data and every quadrant in the domain. The two custom functions, called quadrant and particle functions, one which is called for every quadrant and one which is called for every element in the array, operate without reference to each other. In this case, the quadrant function is only used to calculate the boundaries of the current quadrant in question and store these within the *particles_global* datastructure. The quadrant in question does not need to be local, it can be on any part of the domain. This is done so that the calculation needs to be done less often, as the number of quadrants is significantly smaller than the number of particles, but this is just a design choice and can be modified. The quadrant function always returns a positive value such that the control over stopping or continuing iteration is left to the particle function. It is the same both for particle-cell and cell-particle relationships.

The particle function is different depending on which step of the benchmark is being done. In the case where for every particle the quadrant in which it resides needs to be found, such as for finding the process on which a particle resides after being generated, this is referred to as a particle-cell search in this section for continuity. In the other case, when for a certain quadrant all particles within it need to be found, this is referred to as a cell-particle search in this section.

The particle-cell search function works by first determining if a certain particle exists on the quadrant that is currently being analysed. If this is not the case the iteration can be continued, but if it is the case then we have found a match and therefore just need to determine which process the quadrant in question resides on. Usually this quadrant is only on one process and the iteration can stop, but in some scenarios the quadrant may be split between multiple processes in which case the iteration needs to continue.

The cell-particle search works in a similar way, first determining if the particle in question is contained within the current quadrant. If this is the case, the determination needs to be made if the current quadrant is indeed a leaf quadrant as non-leaf quadrants do not present a quadrant data object and therefore the particle cannot be saved to it. If the quadrant is a leaf quadrant first a check is made to ensure that it is indeed on the local process, which should always be the case but for error management this is done explicitly, and then the particle number is appended to the list of particles on this quadrant and the iteration is continued.

The limiting factor here is the fact that as mentioned earlier, quadrant data is only available for quadrants which are local to the process. This is the reason why the two searches are split, one process finding the quadrant on which its particle resides cannot communicate to the owner process of that quadrant the particle number of that particle as the particles are also locally numbered, in addition to simply not having access to that memory space.

3.8 Communication Strategies

Communication in this library is based on the MPI library. It is needed as through the particle Wigner simulation, particles move between the domains of processes and will need to move between the processes as they do. Communication is a central part of the scaling of a benchmark when going from a single node to multiple nodes as the communication happens not within a single CPU but instead across CPUs. This can lead to significant slowdown in the communication, but is highly dependent on the setup of the system that this benchmark is performed on. This benchmark is performed on the VSC5 cluster described in section 2.3. This cluster has a number of performance indicators, such as its throughput (200Gb/s), latency and buffer size. Each of these performance indicators may impact the performance of the communication step in the benchmark, however finding the specific limitation is difficult and not explored in this work.

The most straightforward option for communicating particles between processes is to simply send all particles to their respective targets in one batch. To achieve this, each process loops through its particles and copies them into one sending vector per target process. These vectors can then be sent from the process to each other process in the system, if there are any particles which need to be sent to that other process.

In order for communication to happen in MPI, both the sending and receiving processes need to know what size the sending/receiving message will be. To achieve this, once each process has filled up their sending vectors a communication round happens where each process sends to every other process an integer which contains how many particles the sender wants to send to the receiver. Now that every process knows how many particles it will receive from all other processes, they first send all the particle sending vectors in a non-blocking fashion before receiving a vector from every other processor (except those which do not have any to send) in a blocking fashion. This one-shot communication method is shown in figure 11.

Executing communication based off a single call per vector can lead to some very large messages being sent, in the case of the SoA structure the length of this message can even surpass the maximum size of an unsigned integer and therefore needs extra handling. These large message sizes can lead to inefficiencies due to memory saturation or cache pollution [23]. In order to test against this, a round based communication strategy was also devised which divides these large vectors into smaller batches which are then individually communicated and put back together on the target process. The round based communication strategy will be referred to as round based communication, and the other as one-shot for clarity in the rest of the thesis.

This round based communication needs a fixed round size to be set for all processes for one communication step. For the benchmark this size is set at the start of the benchmark, but this could be dynamically adjusted when needed. Each round goes through a loop where for every other process, the number of particles that still need to be communicated are determined, followed by the communication of n particles where $n = min(roundsize, particlestobecommunicated)$. The number of particles left to be communicated is decreased by this number of particles communicated and if this number is 0 then a flag is set which removes this process from the send communication loop. After a message is sent to every process in the round, a second loop begins where a message is received from every other process.

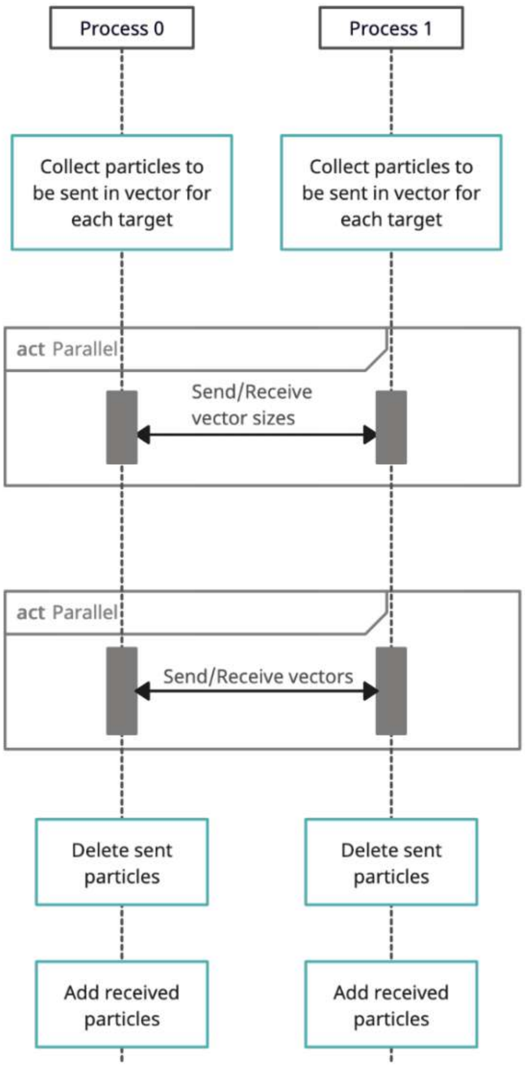


Figure 11: One-shot communication schema shown for two processes.

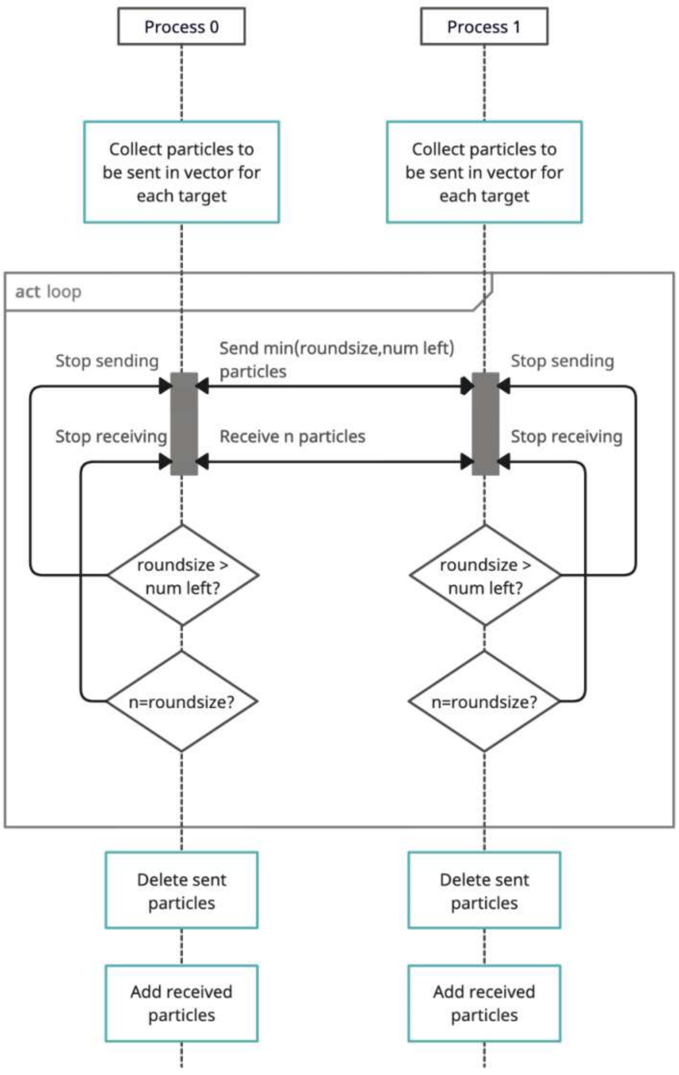


Figure 12: Round based communication shown schematically for two processes.

If this message is smaller than the round size, this means that the partner process has communicated all particles it needed to and therefore a second flag is set which turns off the receive communication from this process. These send and receive loops continue until all processes have set all send and receive flags from all processes, at which point the communication is done. This process is shown schematically in figure 12.

This round based communication does not require the first communication step where processes notify each other of the communication size to be expected. This saves a certain amount of communication load, as only the actual data is communicated. On the other hand, sending more messages using round based communication can introduce some communication overhead as well. Therefore, these two communication setups are benchmarked against each other later in this thesis.

3.9 Benchmark Design

In a complete particle Wigner simulation the most important parts performance wise are those which need to be executed at every timestep. Within each timestep, the particles will move in some way possibly even move across cell boundaries and into the domain of different processes. In the particle Wigner simulation the particle-cell and cell-particle relationships will be required and there may be some refinement of the mesh or repartitioning of the domain. There may also be some particle generation or deletion events.

Simplified, the benchmark takes the approach laid out in figure 13

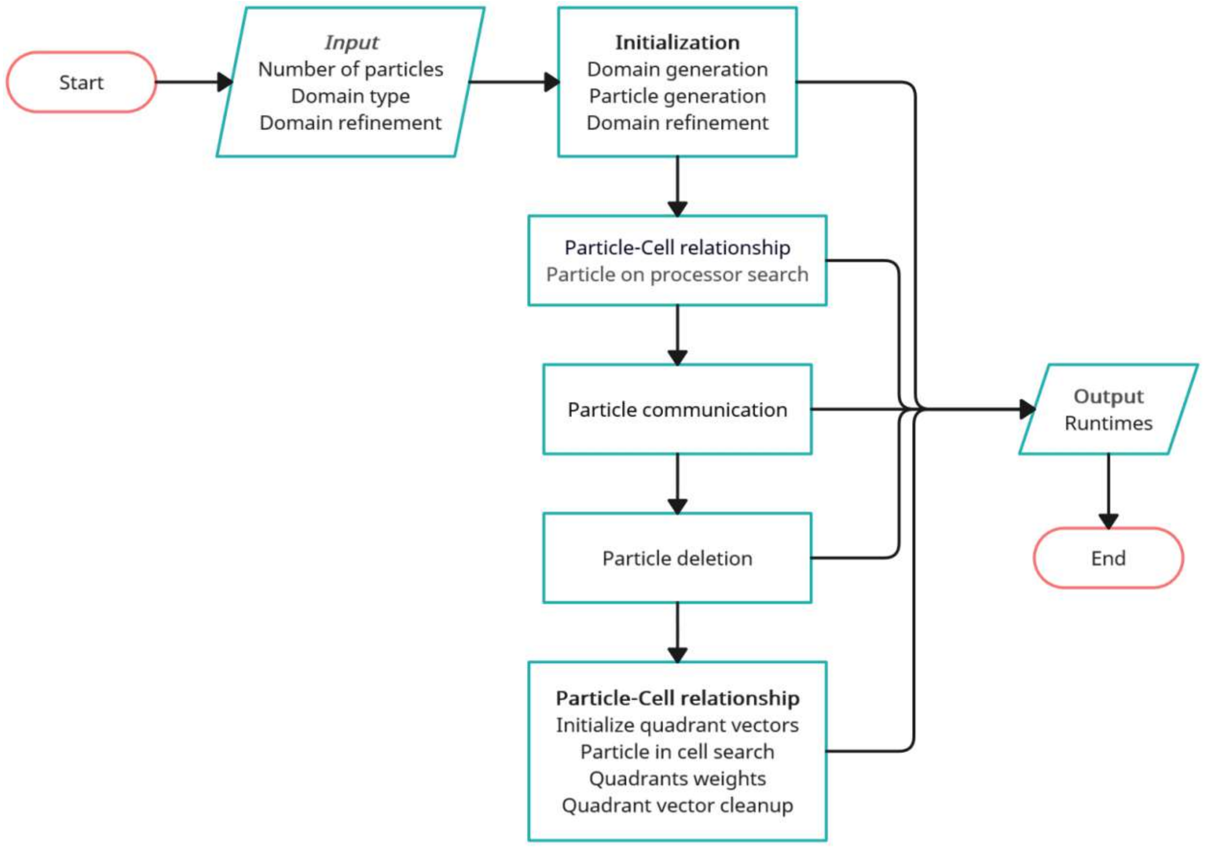


Figure 13: Diagram of the benchmark

In order to cover as many of these types of operations as possible, the benchmark was designed to first generate a determined number of particles on each process, which are generated across the whole domain not just the local domain of the process. Then a search is performed for each particle to determine the process on whose domain it resides, which mirrors the search needed at every timestep after particles have moved and may need to be redistributed among the processes. Then a communication round is conducted where these particles are actually moved between processes to their target domains. The deletion of the particles that have left a certain process is kept as a separate step from the communication itself, as the communication step already disables the particles in question and one may choose to have this deletion step occur less frequently for performance reasons. This allows the deletion step to be separate in the benchmark and provides an almost worst case scenario for benchmarks with high core counts, as here the number of particles that are generated locally but need to move to a different process and therefore be deleted approaches the number of particles generated. After this communication and deletion step, the timestep and generation/deletion events should be done and all particles are at the process in whose domain they reside. This allows for a search to be done where every quadrant obtains a list of all particles that are within its domain. In order to accomplish this however, referring back to the discussion on quadrant level data, the vectors holding this information need to be initialised first, so this step is included in the benchmark. Once the vectors are initialised, this cell-particle relationship search is performed, each leaf cell of the mesh gets a list of particles contained within it and the length of said vector as a weight. This weight is useful for determining if there needs to be a refinement or coarsening in the mesh at some point, as well as a re-partitioning of the domain to better balance it across processes. For this benchmark a re-partitioning of the domain was chosen, so no refinement will be done within the benchmark.

4 Performance Analysis

There are a number of ways to analyse the performance of code execution, including measuring the runtime of the code, the CPU usage during execution and other performance measures. The runtime of the code can be measured in one of two ways, using either the wall time or the CPU clock time. The wall time refers to the time spent between two different execution calls within the code while the CPU clock time refers to the actual time the CPU was busy between these two events. Especially in a distributed system like this there can be significant differences between these two times. The CPU clock time was chosen as a measure for this benchmark as it is the actual real world time that elapses during execution, so it determines how long the compute ressources are bound and the time until the next part of the code can be executed. The wall time is more useful when determining how efficient the compute ressources are used, but that is not the goal of this performance analysis.

In order to measure the performance of the benchmark, every step within the benchmark was timed. This timing was done by synchronising all processes using $MPI\_Barrier$, then on every process saving the time before execution using MPI_Wtime, then calling the function in question, once again synchronising and then taking the difference from the previously saved time to the current time. Therefore all processes synchronise before and after calling of the function in question, meaning the runtime will be within a very small margin across all processes.

The choice was made to run the benchmarking in this way instead of gathering the runtimes of each individual process without synchronising a second time and then aggregating this data for two reasons. The primary one is that the objective of the benchmarking is to find bottlenecks and performance problems, and as this is effectively always taking the worst possible runtime for each step, these performance problems would be more prominent and be unable to be masked by any averaging or other operation done when taking process by process runtimes. The secondary reason is that the functions called in the benchmark typically rely on the previous ones to be completed before continuing onwards, so the choice was made to break them up individually.

4.1 Particle Generation

There were a number of different particle generation strategies laid out in section 3.6 and two of these are now benchmarked against each other. A master-worker generation is the most straightforward way to generate particles as the master process can apply the Monte Carlo method to the generating distribution function without needing to account for the part of the domain these particles are generated on, however

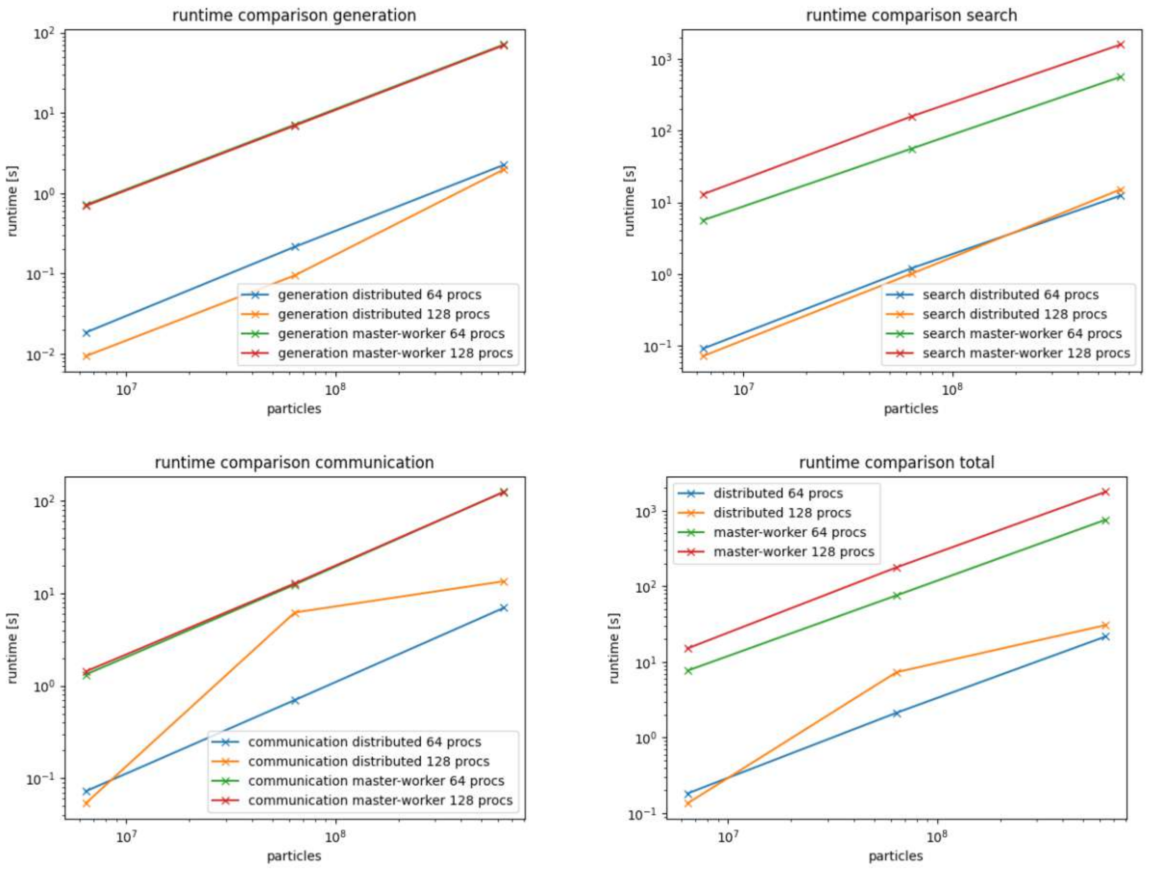


Figure 14: Particle generation methods runtimes in different parts of the benchmark for uniform and non-uniform distributions (figure 10).

as there is only one process generating particles it limits the performance of the generation. As there is only one process generating particles, there is also only one process which needs to do a search to determine which process these particles should be communicated to and then communicate the generated particles to every other process which may be slow in this communication step.

The alternative distributed approach is to allow each process to generate particles across the entire domain, but only generate as many particles as needed on each process such that the entire particle Wigner simulation contains the total number of particles which need to be generated. This means that each process needs to generate less particles, which they can do in parallel to reduce the overall runtime. Then each process finds for each of the generated particles which process's domain this particle resides on and then communicates these particles.

The particle generation, search and communication steps are benchmarked for different particle and process counts and then compared in figure 14. All three of these are needed in the generation process, so a comparison of the total runtime of these three steps is also included.

From figure 14 it can clearly be seen that the distributed way of generating particles performs significantly better, between one and two orders of magnitude faster in runtime across the benchmark. Generation times are faster than particle search times by an order of magnitude, and similar to the communication times. Both generation methods scale linearly in runtime as the number of generated particles increases, which is a good result. There is an outlier in the runtime comparison communication figure for the distributed, 128 process uniform distribution mode, this is a consistent occurrence in the benchmark and further explored in section 4.8.

Due to the better runtime performance, the distributed method of particle generation is used for the rest of this benchmark. In a particle Wigner simulation this distributed generation needs to be investigated for feasibility. As the generation time scales linearly with number of particles, the main particle generation load would be done in the setup phase of the simulation, and generation events within the time step loop would need to generate a much lower number of particles and therefore be faster.

4.2 Particle Deletion

The two deletion strategies previously discussed in section 3.6, simply deleting elements in place or swapping them to the end to then shorten the vector, are now benchmarked to show the extent to which performance is impacted by this. This benchmark is independent of the number of quadrants or processes, and independent of the specific runtimes achieved and therefore run on only a single process. It is meant to show only how the runtime scales with the number of particles which need to be deleted. The results of the benchmark are shown in figure 15

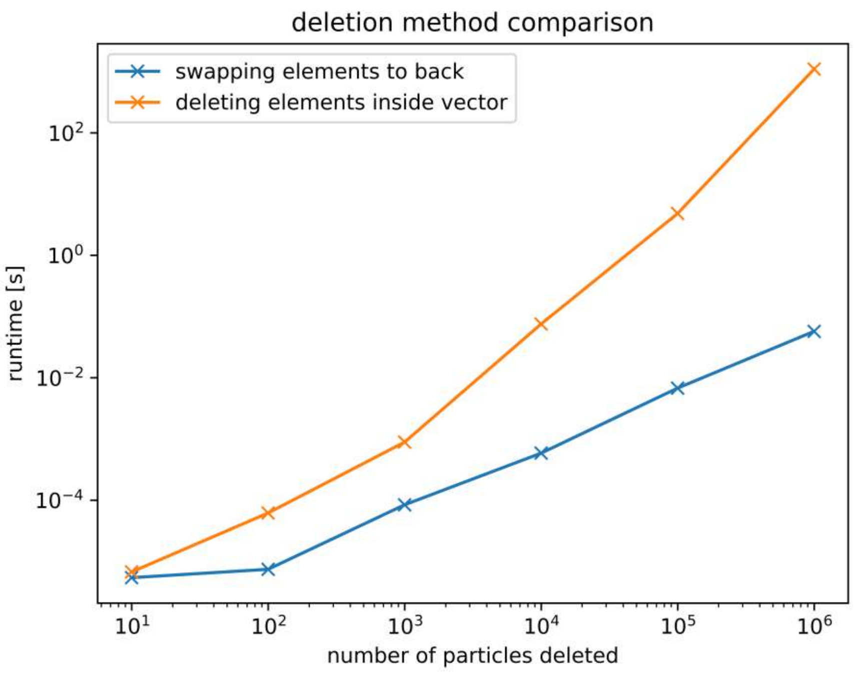


Figure 15: Deletion methods benchmarked against each other.

Shown here in figure 15 is that the method of swapping elements to the back of the array and then deleting them scales linearly with time from the 10^2 number deleted onwards, increasing by four orders of magnitude when the number of deletions goes up by four orders of magnitude. The runtime of the method of deleting elements out of the array individually however increases by around eight orders of magnitude for an increase in number of particles by five orders of magnitude. This increase also is not linear but shows exponential growth seen in the graph, the last two points in the graph going from 10^5 to 10^6 particles increase runtime from 4.8s to 1099s, a 229 fold increase.

Having shown that it is the more efficient method, the method of swapping and reducing the size of the vector is chosen for the benchmarking of the library. In general the deletion step is not the most impactful for performance, as this step is not even visible in most breakdowns of the runtime such as figures 19, 22 and 28.

4.3 Data Storage Options

The performance of the two different memory layouts of the particle data do not show a large difference in performance in the uniform case for single node runs, as can be seen in figure 16. In that figure and all following figures, multi-node runs are marked by a shading of the plot with different colors for single (clear), dual (blue), quad (red) and octa (green) node segments. Those datapoints marked with an X which lie on the border between two shaded areas are part of the left area, as this is the maximum number of processes possible within the smaller node count. The performance difference between them at all benchmarked particle counts differ by no more than 20%, with SoA typically performing slightly better. In the uniform, multi node case, the performance of SoA is generally better than that of AoS by a margin of roughly 20% and both show a similar jump around 10^8 particles, which is an effect that will be discussed in section 4.8. So in the uniform case, the performance difference is not large but favors SoA. In the non-uniform case, the performance advantage flips and AoS has slightly better performance for all but two particle counts in the multinode case.

The performance differences between SoA and AoS are inconsistent between the uniform and non-uniform case and do not allow for a general conclusion that one or the other is performing better. In general the particle Wigner simulation would likely have a non-uniform distribution, however the non-uniform setup in the benchmark is an extreme case and should not be present in an actual simulation.

In the uniform case SoA is better in all situations except two (maximum and minimum particle counts, 64 processes), therefore it can be concluded that SoA should perform better in most load balanced situations. They are also generally small in difference, producing runtimes within 50% of each other. These performance differences may or may not outweigh the performance advantages or disadvantages of the chosen datastructure within the particle Wigner simulation itself. One comparison of these two datastructures was made in [24], where performance was found to be better in every scenario for SoA type datastructures, although the performance difference varies across scenarios. For the remainder of the benchmarks, the SoA method of data storage was chosen as it was found to be the better method for a Monte Carlo Particle Wigner simulation in [24] and provides better performance in the uniform case. A reminder here that the non-uniform case is used to represent a non-load balanced state, while the uniform case represents one which is load balanced. A very non-uniform particle distribution can still be load balanced using the *p4est_partition* function, as described in section 3.3.

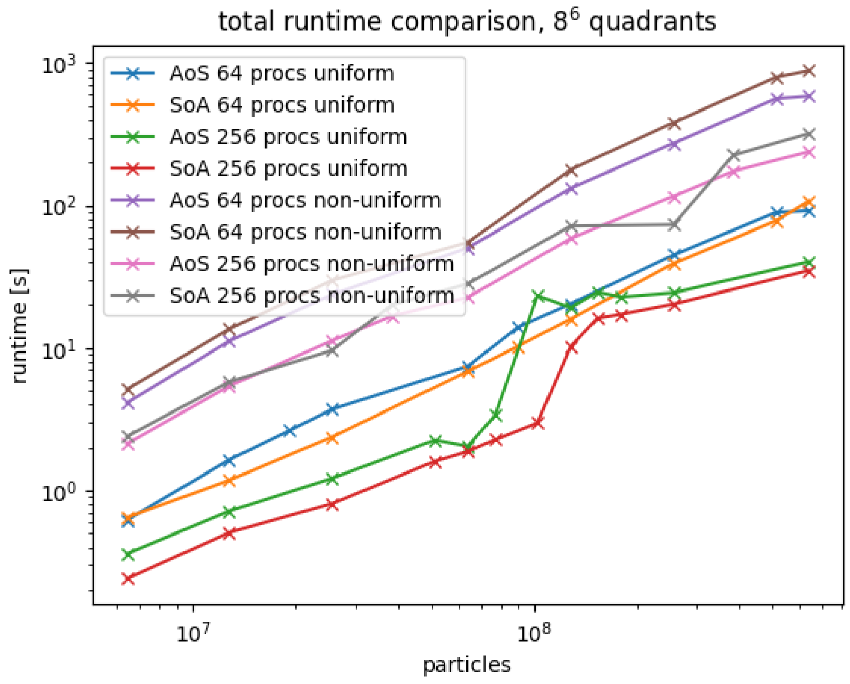


Figure 16: Struct of Arrays vs Array of Structs, total runtime comparison.

4.4 Communication Strategies

The two communication strategies, round based and one-shot communication outlined in section 3.8 are now benchmarked against each other in figure 17.

The performance shown in figure 17 is quite different between the different particle counts. In the smaller particle counts, the round based communication is worse when the roundsize approaches the number of particles per process, but performs similarly when the roundsize is below 10% the number of particles per process. There is a local minimum when the roundsize is around 1% of the number of particles per process, where the communication is consistently better for all process counts.

For the larger particle count benchmarks, this trend however does not hold as the communication behaves very erratically. In both lower plots of figure 17 there is an outlier, with 128 and 512 particles respectively, for which the communication when using a very small round size is much faster by an order of magnitude.

This inconsistent behaviour depending on the particle count means that there is no way to make a general best choice for the round size, and therefore this round based communication is not used in the rest of the benchmark.

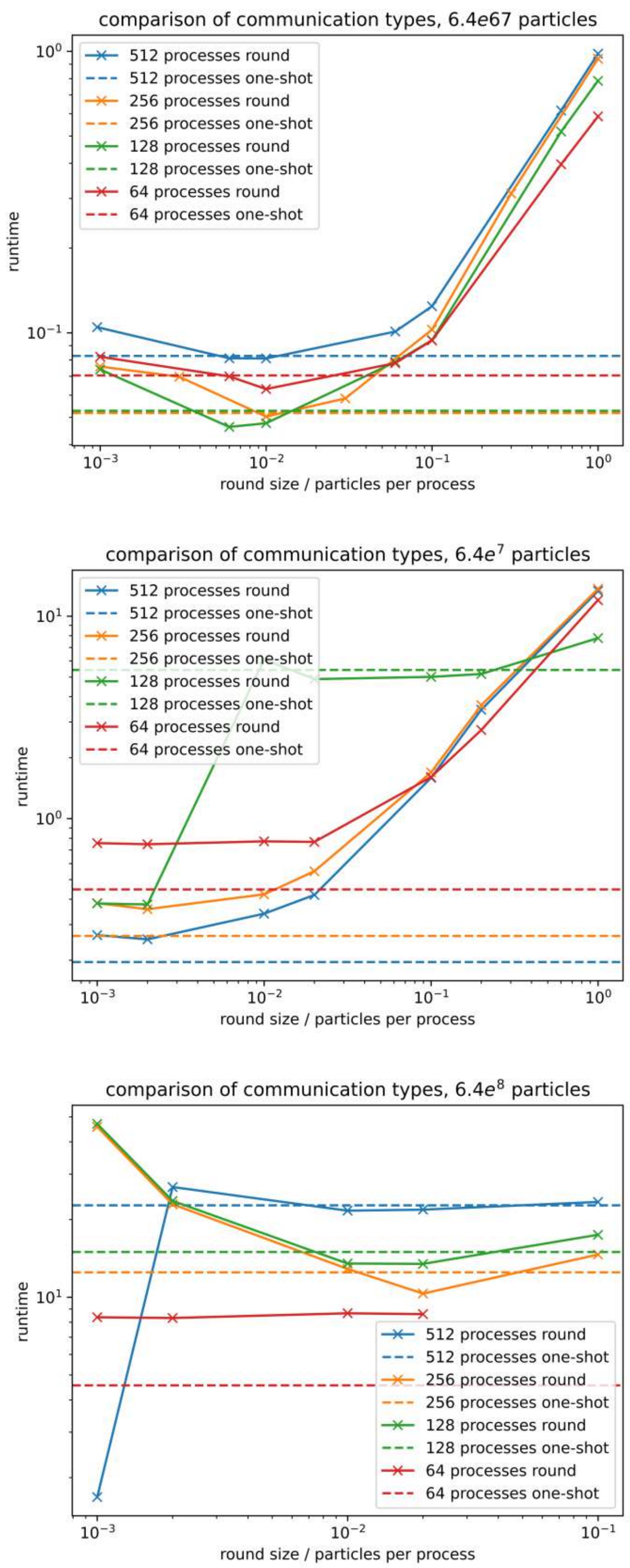


Figure 17: Round vs one-shot communication

4.5 Weak Scaling Performance

An important measure of the performance of a program is the weak scaling. Weak scaling is defined by Gustafson [25] as

$$scaledspeedup(S_s) = s + p \cdot N$$

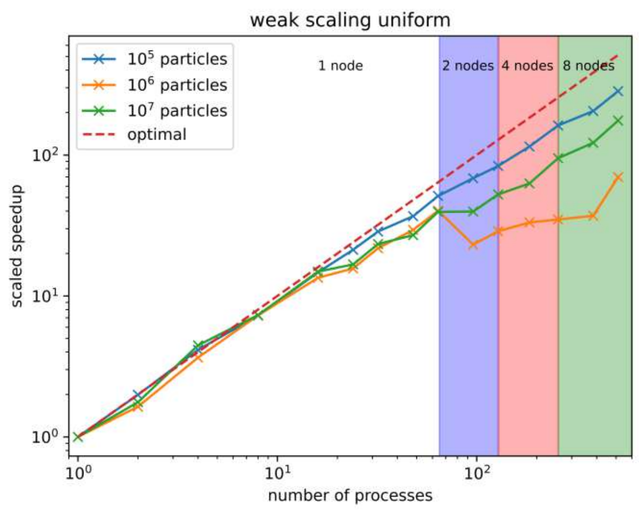
where s is the part of the code that needs to run serially, N is the number of processes and p is the part of the code that can run in parallel. While strong scaling refers to the efficiency of increasing process count with the same problem size, weak scaling instead measures the efficiency of increasing the problem size in parallel with the process count. Figure 18 shows the scaled speedup of the benchmark, with the dotted line representing a perfect speedup with no losses from increased process count. For these figures, the scaled speedup is calculated by taking the runtime of any execution, dividing it by the runtime of the single process execution and multiplying with the amount of processes used, which is also the factor by which the workload increased compared to single process execution. The calculation for this speedup is:

$$S_s = \frac{t_1 * n}{t_n}$$

where n is the number of cores, and t_n is the runtime when running with n cores.

The number of particles in the figures here is the number of particles per process, unlike in other figures in this work where the total number of particles is noted.

Shown in figure 18 is the scaled speedup of the benchmark. In the uniform case, the plot stays very close to the optimal line until about 30 processes, at which point there is a slight slowdown. For the smaller number of particles, there is a quite significant drop off going from 64 to 96 processes, with the larger particle part also showing a slight dropoff. This is explained by the fact that 64 processes are run on only one node, while for 96 processes at least two nodes are needed which significantly increases communication times. This phenomenon can be seen very clearly in the figure 19, which shows the runtimes of each part of the benchmark as a stacked chart.



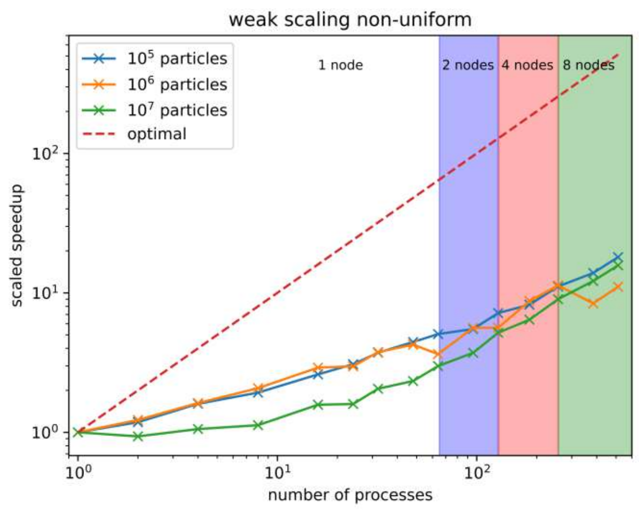


Figure 18: Weak scaling of the benchmark.

Shown in figure 19 is the significant increase in communication times going from 64 cores to 96, which is explained by the fact that communication between multiple nodes is more expensive than communication within one node. There are some trends in the plot for 10^7 to be further discussed as well. In comparison to the smaller particle count, the total runtime bounces around a bit more, especially in the particle on process search function. This is the function that initially determines for each particle in which process's domain it is located. The reason why this function increases in runtime is not entirely clear, as it only takes the freshly generated particles and searches for each of them, so the workload should be perfectly evenly distributed. However since the generation of particles is completely random and the runtime of the slowest process is used for these benchmarks, there is a certain amount of variance to be expected.

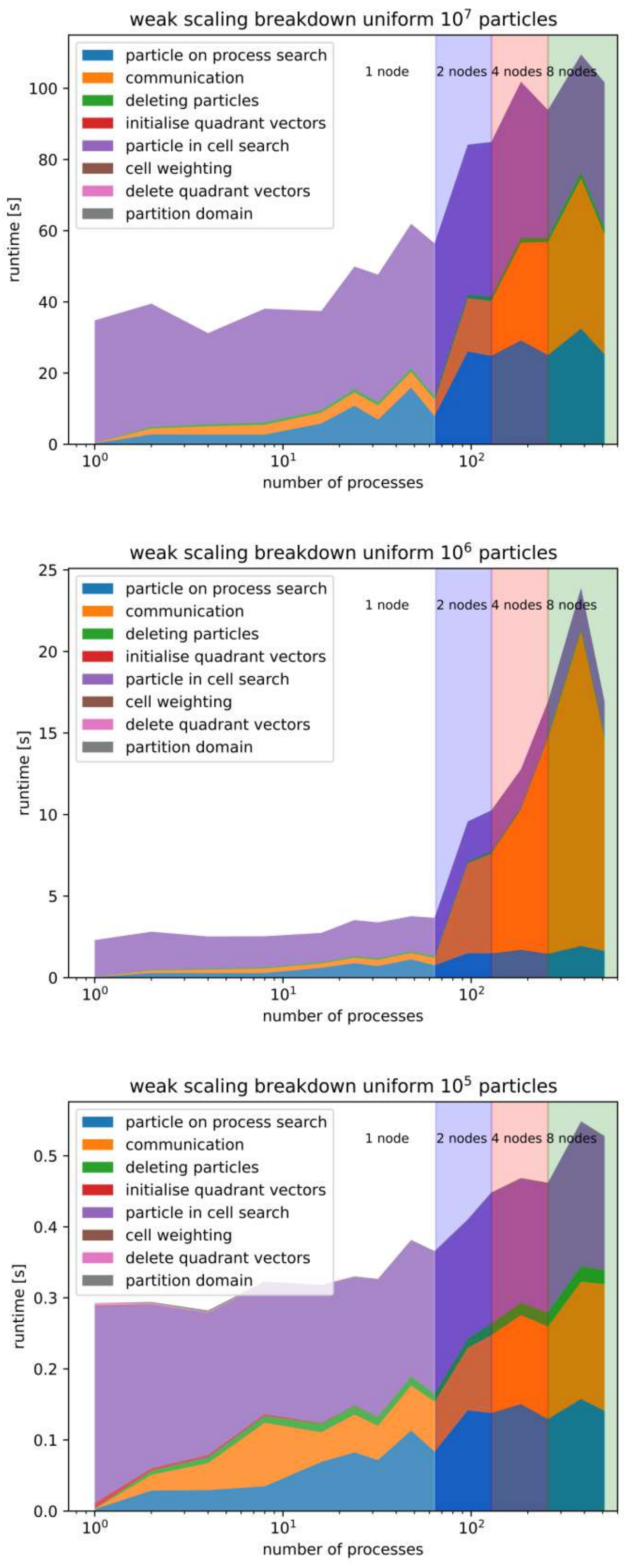


Figure 19: Runtime breakdown of weak scaling benchmark, uniform.

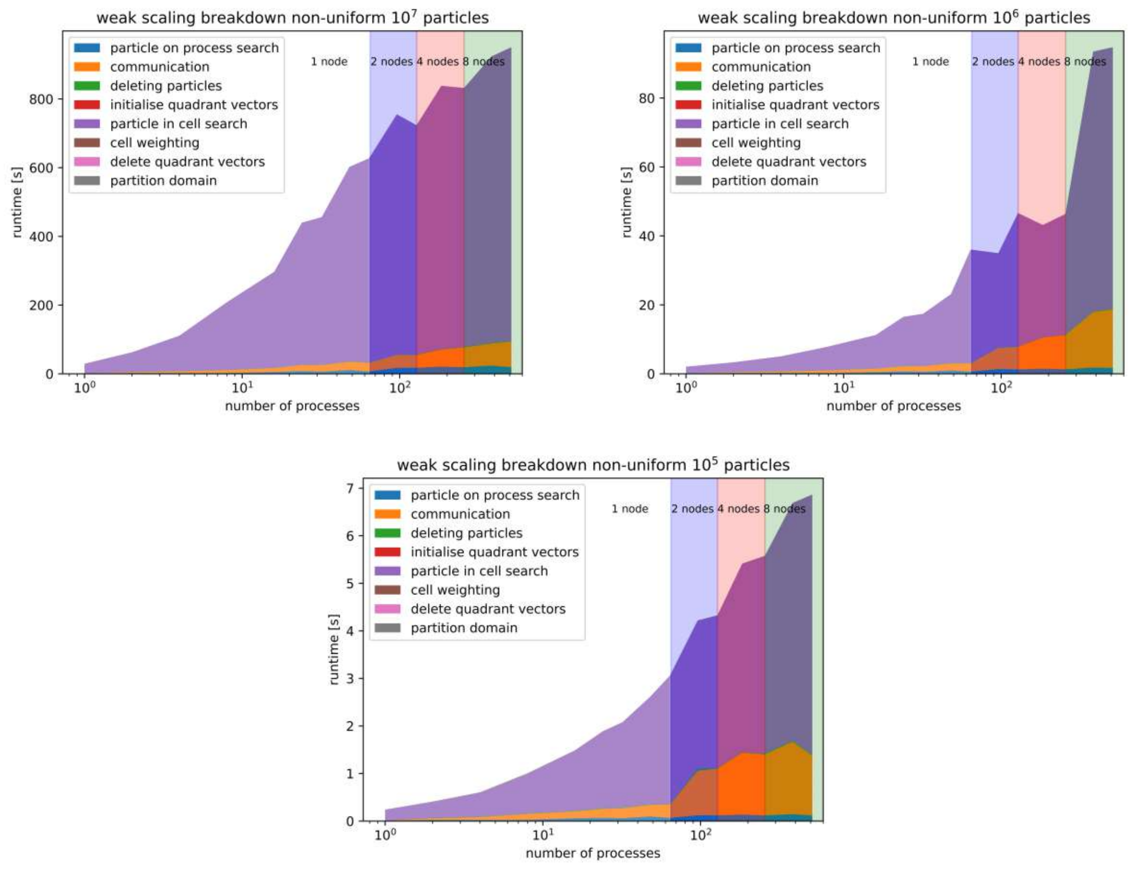


Figure 20: Runtime breakdown of weak scaling benchmark, non-uniform.

In the non-uniform case, weak scaling is terrible from even just 2 processes onwards, and the scaled speedup barely manages to get above a factor of ten when using 512 total processes. Especially for the smaller number of particles, the scaled speedup stays below 1 for a majority of single-node runs, which means that it would be more efficient to run on a single process with the larger problem size. This is rather alarming, but the explanation for this behaviour is quite straightforward: the non-uniform distribution is much too punishing for a larger system as the number of particles per process is very different, in the 64 process 10^7 particle case process 0 has $1.35 \cdot 10^8$ particles while process 63 has $2.5 \cdot 10^4$. With this kind of distribution there are obviously going to be some processes that take a lot longer to execute each part of the benchmark than others, which means as we need to wait for them all to finish parts of their execution, the total runtime of the benchmark suffers heavily. This is not a scenario which should occur when attempting to use this library for implementing a custom particle Wigner simulator, but if this problem shows up in some part of the simulation where the distribution is uneven there may be a heavy cost associated with it.

When implementing a particle Wigner simulation using p4est, one may want to look at how one is generating and refining their mesh before the generation of particles, or may want to skip this step all together. This search function is generally used for finding which particles belong in every cell within the entire domain, but this information is not strictly necessary if the ultimate goal is just to determine where one should refine the mesh and/or partition the mesh among processes to ensure an even distribution. Both the refinement and partition steps just require a function that either decides to refine or not, in the refinement case, or determines the weight of a cell which then informs the partitioning of the domain. Since the particles are generated randomly as part of a distribution over some part of the domain, the weighting and refinement functions could be written in such a way that they mirror this particle distribution only using the position and size of each cell rather than finding individual particles within it. This would result in a not perfect, but relatively good solution which takes orders of magnitude less time.

4.6 Strong Scaling Performance

Another measure of the parallel performance of a program is the strong scaling. For strong scaling, the total workload of the problem is kept the same while the number of processes assigned to it are increased. This is also covered by Amdahl's law which can be formulated as

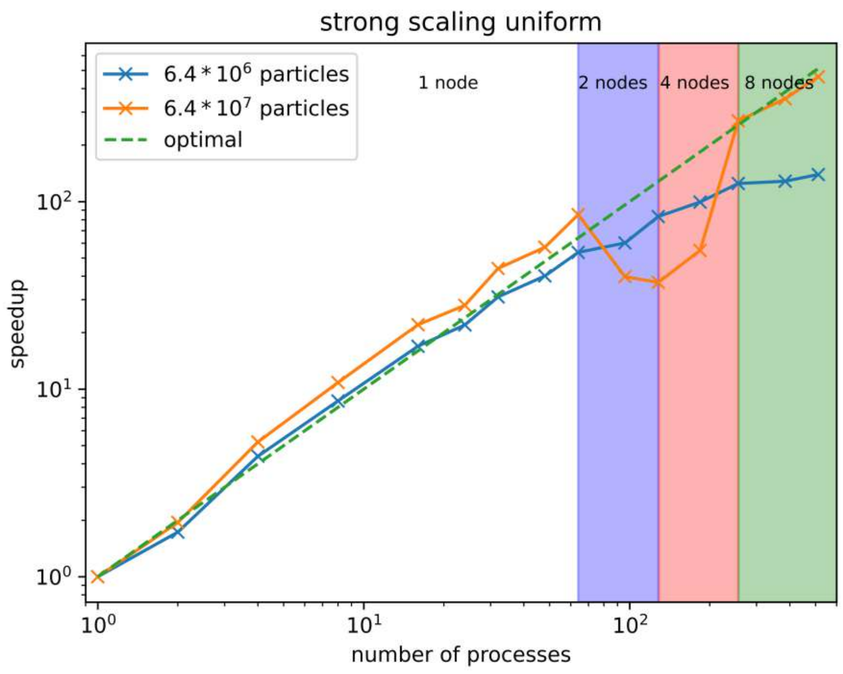
$$speedup = \frac{1}{s + \frac{p}{N}}$$

where s is the serial part of the program that cannot be parallelized and p is the parallelizeable part, with N being the number of processes.

Strong scaling is defined as

$$speedup = \frac{runtime_1}{runtime_n}$$

and a corresponding benchmark scenario is shown in figure 21.



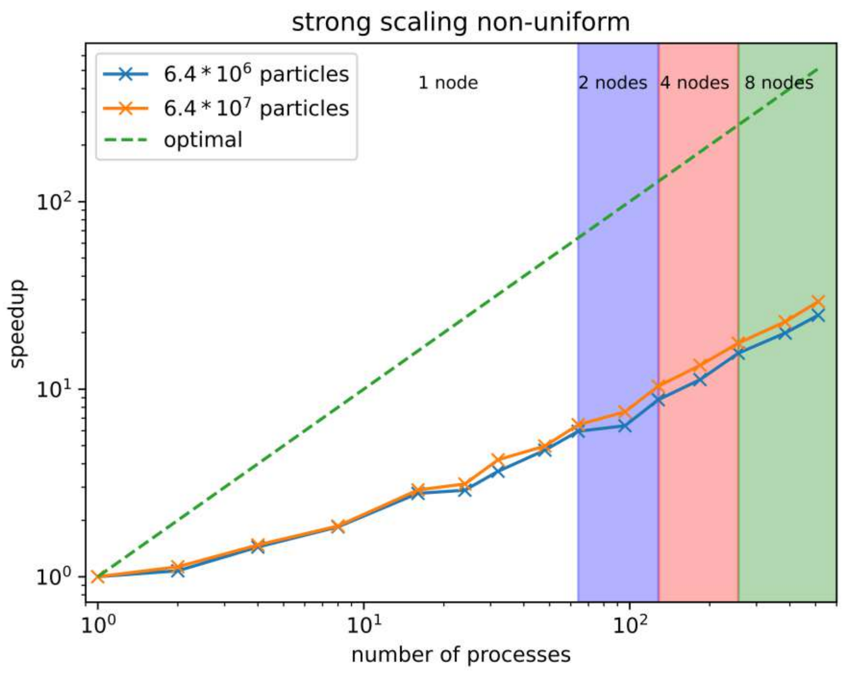


Figure 21: Strong scaling of the benchmark.

Here once again the uniformly distributed version scales quite well, with some expected drop-off when the number of nodes increases from one to two as the communication time increases between nodes. In the non-uniform case the scaling once again is a lot weaker, but the speedup does increase more significantly than in the weak scaling case as the number of processes increases.

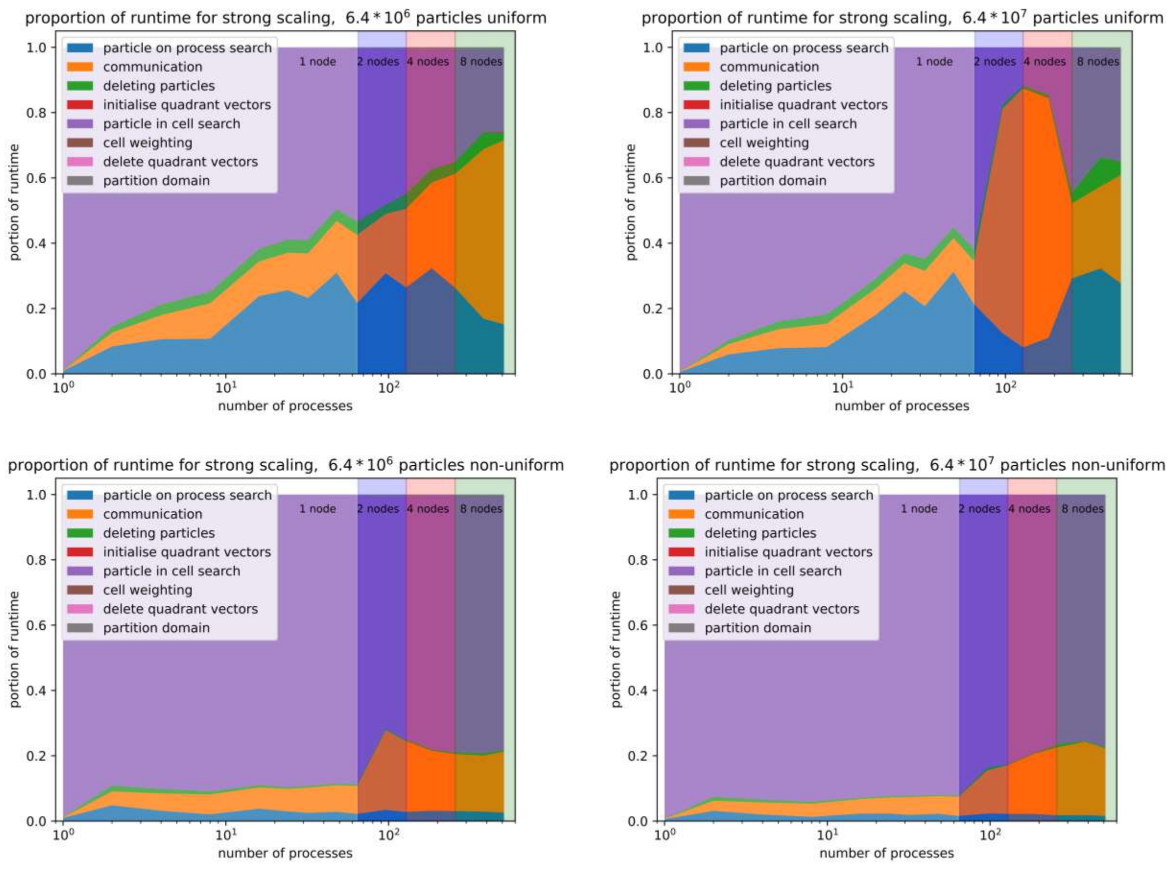


Figure 22: Proportional runtime breakdown of strong scaling benchmark.

The breakdown of the runtimes of this strong scaling benchmark is shown in figure 22 proportionally as the runtimes continually decrease as the number of processes increases. In the uniform case, the particle in cell search continually declines in proportional runtime, as the initial particle on process search increases somewhat in proportional runtime. This is all still within the part of figure 21 where the scaling is in line with the optimum. There is a decrease in strong scaling when going from one node to multiple nodes, which is to be expected but not immediately obvious in the breakdown for $6.4 \cdot 10^6$ particles, and completely dominant for $6.4 \cdot 10^7$ particles. These two cases need to be treated differently, as for $6.4 \cdot 10^6$ particles as the number of cores increases further and the strong scaling continues to dip, the breakdown shows that the communication load continues to increase and eat up more and more of the runtime of the benchmark, until about half of the runtime is dedicated only to communication. For the $6.4 \cdot 10^7$ particles case, the communication load explodes for core counts 96, 128 and 184 before dropping back down to normal levels for higher core counts. This corresponds to the drop-off in figure 21 for these same core and particle count combinations, and shows a communication slowdown is happening.

In the non-uniform case, the breakdown does not show much to explain why the strong scaling is so bad at the beginning and then slowly catches up as the number of processes increases. There is a sharp increase in communication time when going from one node to multiple nodes, which is also visible in figure 21 but this does not continue to increase like in the uniform case. This bad strong scaling can be better explained by the setup of the benchmark and the definition of strong scaling itself. As the benchmark is set up for the non-uniform case, the particles are pushed into the corner at [0,0] with very few in other locations while the processes are evenly spread across the domain. The benchmark also uses the particle in cell search function, which means that every process searches through every cell and particle it has locally on its domain. This search is effectively not parallelizeable, the way through which it is parallelized is through the domain being split across multiple processes. The benchmark also only takes into account the maximum runtime of any process, as this would be the bottleneck for any real scenario. Now, when the domain is evenly split between processes, but the particles are distributed such that certain parts of the domain have much more load, and this increased load is effectively not parallelizeable, the strong scaling will be quite bad at the beginning. As the number of processes increases, the parts of the domain which contain a lot of particles will start to be split between multiple processes and the previously not parallelizeable parts will start to be parallelized, leading to better strong scaling numbers although this will obviously not reach the optimal values.

4.7 Throughput Performance

In this section, another way to measure performance, so-called throughput performance, is evaluated. It measures the number of particles processed per second. This measure is relevant to the particle Wigner simulation as the number of particles can vary between time steps and is limited to a certain upper bound defined in the simulation setup. Finding a throughput increase or drop-off beyond a certain number of particles would be useful to inform this simulation setup. This throughput can be used to identify the parts of the benchmark that are limiting performance, as the overall throughput cannot be better than the worst part of the benchmark. Throughput is in this case defined as just the total particle count in the benchmark divided by runtime of the step, not accounting for different steps possibly operating on not all particles such as for communication. In figure 23 the throughput in particles per second is shown for a number of particles, using $8^6 = 262144$ quadrants.

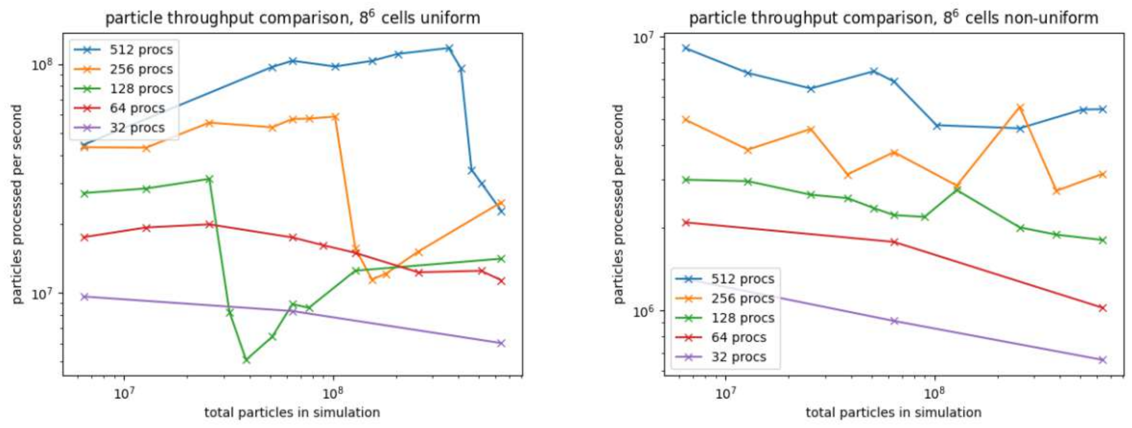


Figure 23: Particle throughput for different problem sizes.

While generally figure 23 shows relatively strong scaling with increasing number of particles, where the throughput is only decreased by roughly a factor of two when increasing the number of particles by two orders of magnitude, this consistent scaling only applies to a single node (32 and 64 cores) benchmark. This behaviour is quite consistent for both uniform and non-uniform distributions, where the lines are roughly parallel. There is the issue with the communication slowdown for multi-node benchmarks, which is further shown in figure 24, top left. As the communication throughput massively declines when hitting a certain communication load, the overall throughput goes down to match that. On top in figure 24 is the first communication after the particles were generated, and on the bottom is the second communication of particles post re-partitioning of the domain. The second communication step shows a much larger throughput, although in the uniform case the communication load is very low as the partitioning changes very little in the domain.

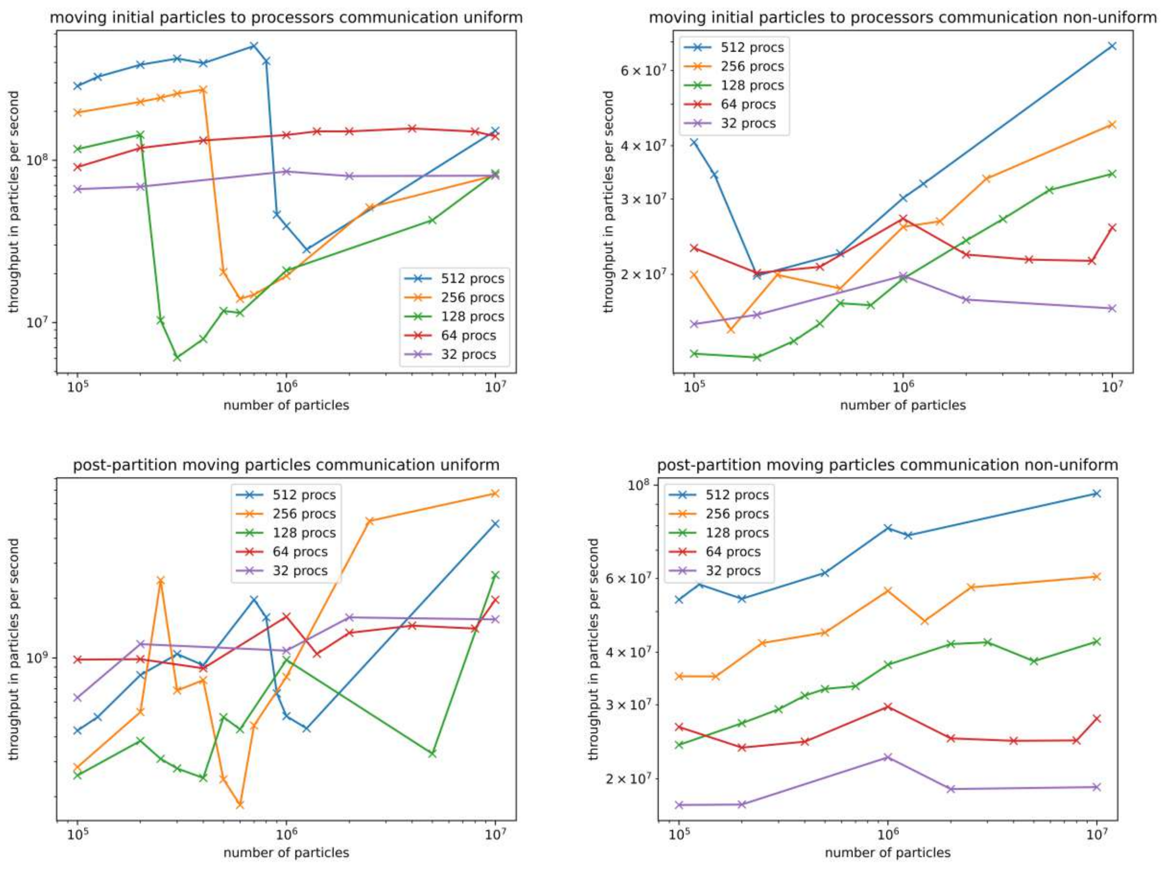


Figure 24: Throughput of two communication steps.

4.8 Communication Bottleneck

In the testing of the benchmark for different situation, an edge case was found which will show up in different results like figures 16, 21 and 23. This happens when the communication load increases over a certain value at which point the communication time jumps from below one second ($< 0.2s$, $< 0.4s$

and $< 0.8s$ for 2,4 and 8 nodes respectively) to a minimum of 6 seconds. The problem size at which this jump occurs is only obvious in the uniform case and dependent on the number of nodes, with $2.56 \cdot 10^7$ particles for 2 nodes, $1.28 \cdot 10^8$ for 4 nodes and $4.61 \cdot 10^8$ for 8 nodes producing this slowdown. This produces a jump in the respective values being analysed for this specific problem size and node count combination. This effect is especially prominent in figure 22, top right plot. Here we see the proportion of the benchmarks runtime dedicated to each function. Going from one to two nodes in this situation, with a total problem size of $6.4 \cdot 10^7$ particles regardless of number of processes, puts this problem above the previously mentioned $2.66 \cdot 10^7$ critical problem size for 2 nodes, but below the $1.28 \cdot 10^8$ number for 4 nodes. Therefore one should expect the benchmark run on two nodes to produce very slow communication compared to 4 and higher node counts, which is the actual result both in figures 22(top right) and 21(left). The communication slowdown does not occur at all for benchmarks running on only one node, as the communication here only occurs within the node and does not rely on the communication layer between nodes, where the bottleneck occurs.

This communication slowdown is related to the high communication load that this benchmark creates by design, as mentioned in the implementation of the communication algorithm. Such a high communication load, where most particles need to be communicated, can be avoided by distributing the generation event in a more elegant way such that particles are generated by a process only in their respective domains.

This communication slowdown does not occur in such a discrete fashion for the non-uniform case. This is due to the fact that the other functions, specifically the particle-cell relationship search functions, already exceed the communication in runtime for the given particle counts and therefore overshadow the increased runtime in communication.

The performance of round based vs one-shot communication was also benchmarked in section 4.4, but no improvement to this bottleneck was found when switching over to round based communication. There were two outliers mentioned there in which the bottleneck appears to have been resolved, but these were individual outliers and did not appear consistently.

4.9 Mesh Refinement

The refinement of the mesh is important to the performance and numerical accuracy of a particle Wigner simulation. As the number of quadrants increases the runtime is expected to increase as well, but the rate at which it increases is relevant to the overall performance of the system. The parts of the benchmark that are most relevant to this are the particle-cell relationship search functions, as well as the quadrant vector init, quadrant vector delete and partition functions. The particle search functions are dependent on both the number of particles as well as the number of quadrants, while the others are purely dependent on the number of quadrants and therefore can be used to show performance of functions which act exclusively on quadrants.

Figure 25 shows how the runtime of the benchmark changes as the number of quadrants in the mesh is increased. In general, the scaling with number of quadrants is much better than the scaling with number of particles. There are roughly seven orders of magnitude increase in the number of quadrants with an increase in runtime by one order of magnitude, while the runtime increases linearly with number of particles. This shows that p4est manages very fine meshes very efficiently.

There are two interesting phenomena to look at in figure 25. One is the increase in runtime as the number of quadrants increase for the lower end of number of particles, and the other is the significant difference in both runtime and scaling of the 64 and 4096 quadrant counts compared to the others, which exhibit similar behaviour to each other.

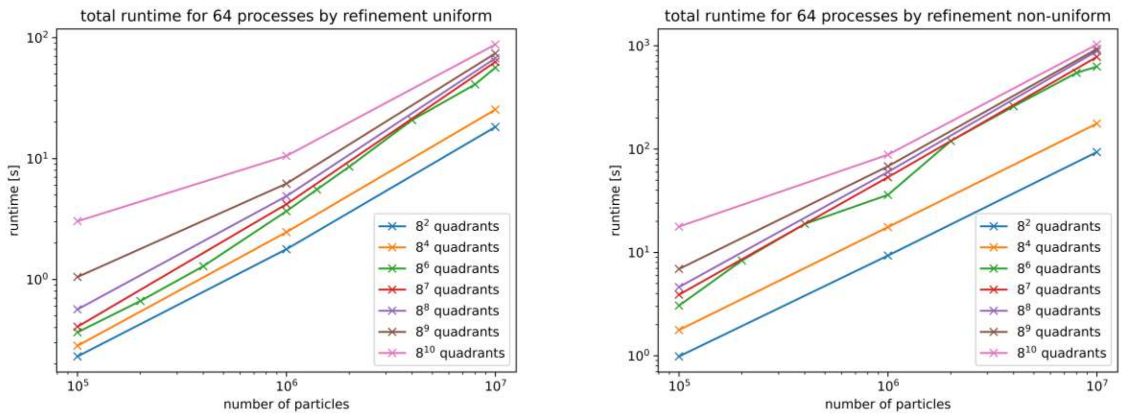


Figure 25: Total runtime of the benchmarking suite with uniformly or non-uniformly distributed particles.

Both phenomena are largely explained by figure 26. For this benchmark, the particle count was radically decreased to $6.4 \cdot 10^4$ to focus solely on the performance impact of having more quadrants. The total runtime plot for this benchmark already shows a pretty clear picture, as the number of quadrants increase the runtime stays relatively flat with only minor increases until the number of quadrants reaches about 10^6, at which point the runtimes increase roughly linearly with the increase in quadrants.

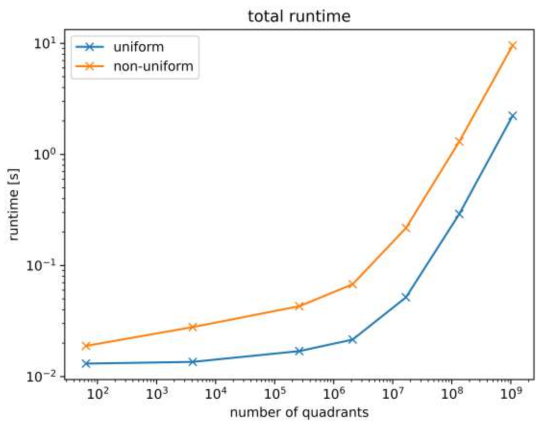


Figure 26: Total runtime of the benchmark over the number of quadrants, 64000 particles, 64 cores.

Breaking down this benchmark into individual components, two of which are shown in figure 27, some parts show no significant changes in runtime with increasing number of quadrants, some like the post-partition particle in cell search show increases in runtime that are sublinear, which does not explain this total runtime plot. The one that stands out here is the partitioning step. This shows a similar shape to the total runtime plot, and coincidentally also produces similar runtime totals to the total runtime. This is one of the limiting factors for this benchmark, the re-partitioning process. This partitioning step is also entirely independent of the number of particles in the benchmark, it only deals with the quadrants.

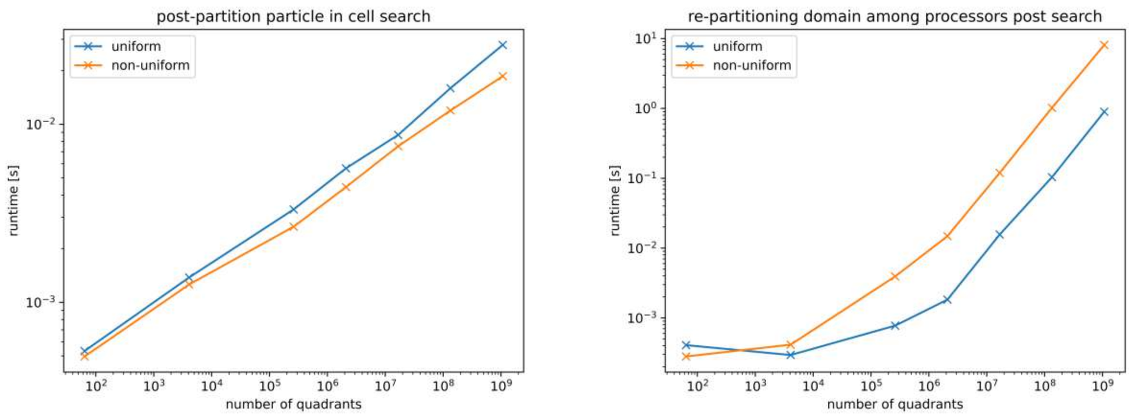


Figure 27: Two individual steps of the benchmark, runtime over number of quadrants, 64000 particles, 64 cores.

Another way to show this increase in runtime due to the increasing number of quadrants is the breakdown of the runtime by function for the same setup shown in figure 28. Here the communication will always take the same amount of time, as it is entirely independent of the number of quadrants, which shows the performance of the other parts of the benchmark scaling with the number of quadrants. As the number of quadrants increases first the search functions increase in runtime and start to be the performance bottleneck, but around 10^6 quadrants the partition, quadrant initialization and quadrant vector delete functions begin to dominate the runtime which is in line with the massive increase in runtime at the same quadrant count in figure 26.

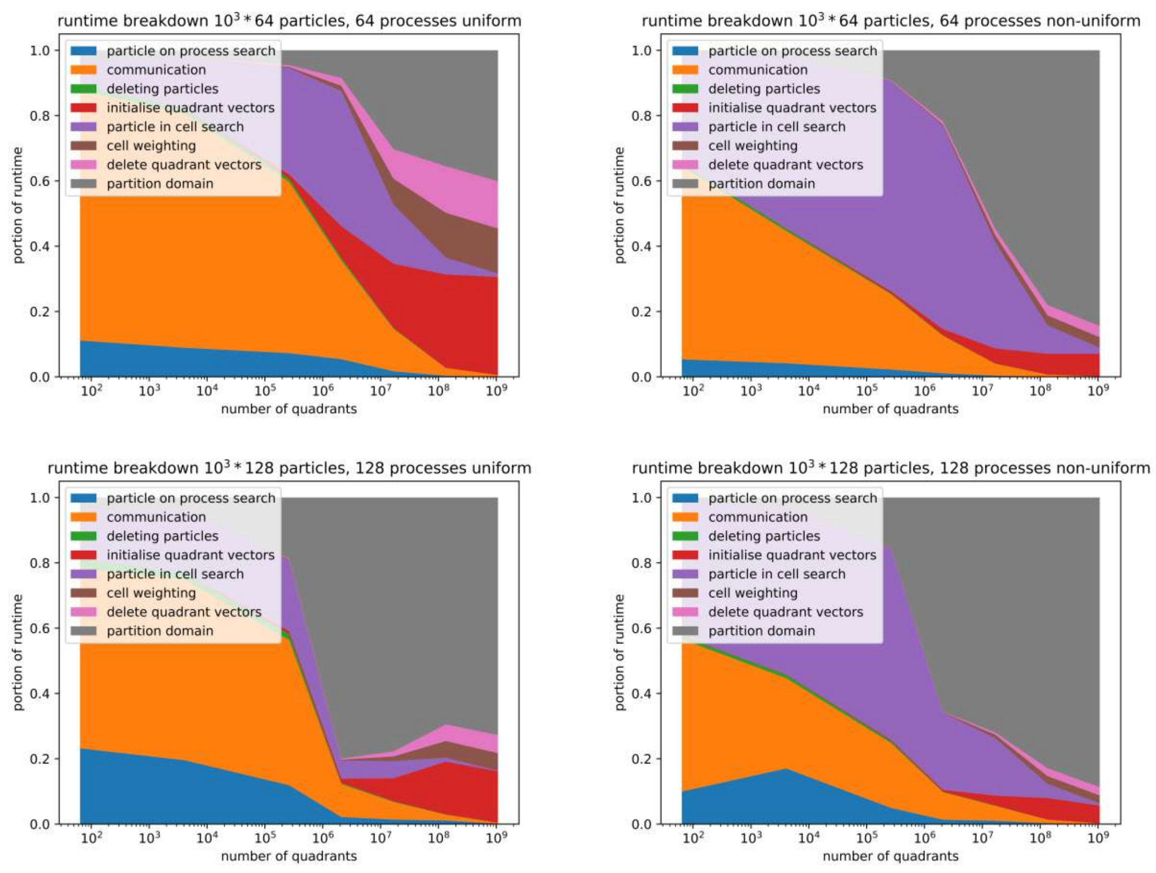


Figure 28: Runtime breakdown of the benchmark for one and two nodes, uniform and non-uniform.

The other phenomenon in figure 26, the higher increased runtime for larger cell and particle numbers over lower cell numbers is due to the particle cell search function. This can be seen on the left in figure 27 where increasing the number of quadrants increases the runtime, but only by less than two orders of magnitude for seven orders of magnitude increase in cell count. This part of the benchmark, the particle cell search, is responsible for the largest part of the runtime in the other typical scenarios when particle counts are much larger than 1000. Since this runtime scales with both the number of quadrants as well as the number of particles, an increase in runtime in the top right corner of the graph in figure 25 is to be expected.

5 Memory usage

In a simulation like the Monte Carlo particle Wigner simulation which has been covered in this thesis, an important aspect of performance is the memory usage of the simulation. The more memory is used within the simulation the more computing resources are required, which increases the cost of the simulation, and the slower the simulation runs as more memory needs to be accessed overall. The memory usage should theoretically be determined by the size of the simulation, how many particles and quadrants exist within it, multiplied with the memory consumed per particle/quadrant. In the application of the simulation however there will be a certain amount of overhead, data stored which is not directly part of the simulation but required to support it, which will have a negative impact on performance. This section aims to determine the amount of memory overhead which is present in the benchmark as well as to give some information on the amount of memory consumed by the p4est library, as this library is not managed by the benchmark itself and therefore needs to be experimentally explored.

In order to determine the amount of memory used in a parallel MPI application, a number of open source libraries exist which track this memory consumption on a per thread basis. In Linux the *top* command is typically used to look at the memory usage of an thread as it displays the amount of memory used, percentage of CPU resources used by the application and the thread ID which can be used to give more direct information. For an MPI application this command displays the wrong information however as it displays the total memory consumption of the application on each thread of the application, so a multi threaded application using four threads which each consume 400 MB of memory would display as four threads each consuming 1.6 GB of memory in *top*. In order to get more concrete information on the memory consumption a number of libraries were considered in the creation of this benchmark, namely Valgrind using Massif, MPIP (MPI Profiling Library), TAU (Tuning and Analysis Utilities) and gperftools. Gperftools was chosen for this benchmark as it is the most straightforward to apply to an existing benchmark, it only requires linking the library in the compilation process while the others require replacing the compiler and therefore have an influence on the MPI library as well. It is missing a certain number of features that the others contain but these are not required for analysing the memory consumption of the benchmark. Gperftools was developed by Google to do analysis on the MPI applications that they developed and released under the BSD License which allows a wide range of uses without licensing issues.

The important feature of gperftools in the analysis of memory consumption within the benchmark is the heap dumping. A heap dump is a snapshot of all objects stored within memory used by the application at a certain point in time. Gperftools allows the user to generate these dumps automatically when a certain trigger is met, such as when a certain amount of total memory is consumed or when memory consumption increases by a certain amount, as well as by manually inserting a command within the application which creates a heap dump at a certain point in the code. These heap dumps are then compiled into a readable format by generating an image which shows the specific execution calls which reserved the memory that is in use as well as the call stack which tracks where in the program these calls were executed from. Each call is represented by a rectangle which has the memory it required displayed as a number, as well as edges connecting these rectangles which also display the memory used. So if a call is made within the main function which creates a vector that requires 200 MB of memory for example, there would be one rectangle representing the main function, annotated as requiring 200 MB of memory, connected by one edge also annotated to be 200 MB to a call to the standard library function which generates the vector, again annotated with 200 MB. It should be noted that gperftools drops connections which are small in size relative to the overall memory footprint in order to reduce clutter. This is shown in the top left of any analysis output image generated by gperftools.

5.1 Particles

In order to benchmark the memory consumption of the particles within a particle Wigner simulation, the impact of the memory consumed by p4est must be reduced as much as possible. In order to achieve this the number of quadrants is reduced to a minimum, however the parallelisation requires at least as many quadrants as processes otherwise parallelisation is not possible. As such, a setup was chosen with a cubic domain which was refined once to create eight quadrants, and then this benchmark is run on eight processes with a large number of particles. This leads to a memory footprint that should be dominated by the memory used by particles, and as gperftools drops certain edges and nodes the memory footprint of the p4est datastructure should not even be present in the output.

One small modification was made for this benchmark by moving the deletion step from outside the communication step to within it, after the communication has occurred so particles can be safely deleted but before the received particles have been added to the datastructure and are still within temporary storage. This is because the deletion step is not required to be individually benchmarked for runtime in this case, and when communication occurs the majority of particles existing on the process are deleted and roughly the same number of new particles are added. Adding these particles before deleting them unnecessarily increases the memory required to store all the data and would be inefficient within an actual particle Wigner simulation, the only reason it is even moved out of the communication step is that it provides a convenient benchmarking opportunity for the deletion step.

The output of such an analysis with 10^7 particles in the benchmark with 8 processes and quadrants is shown in figure 29. This heap is dumped just after communication, when the communicated particles have been deleted but the quadrants have not yet had their containing particles vectors filled up. After those have been populated, the heap is dumped again and shown in figure 30.

As there should be about $1.25 \cdot 10^6$ particles on process 0 in this case, as each process generates that many particles and the distribution is uniform. Each particle has a memory footprint of six doubles and one integer, plus the sc array which is needed and contains one integer per particle. There is additionally one vector within the global datastructure, the *particles_target_processor* vector which stores for each particle what process it should reside on. This information is needed for communication but set in a different function and therefore needs to be global. The same functionality could also have been implemented in a different way such that this *particles_target_processor* vector gets cleaned up after the communication and again allocated before a search is performed. This more memory efficient approach was not deemed necessary for the benchmarking of runtime performance so therefore not included, however it means that an additional integer is stored for each particle. The total memory footprint in this case should be about $1.25 \cdot 10^6 \cdot (6 \cdot 8bytes + 2 \cdot 4bytes) = 7.0 \cdot 10^7 bytes$, equivalent to about 70 Megabytes(MB). The sc array should contain about $1.25 \cdot 10^6 \cdot 4bytes = 5 \cdot 10^6 bytes$ or about 5 MB while the vectors should contain about $1.25 \cdot 10^6 \cdot 52 = 6.5 \cdot 10^7 bytes$ or about 65 MB. The analysis also includes the memory consumption after quadrants vectors have been filled with the particles contained in them. These vectors exist once per quadrant and contain an integer which is the index of the particle in question, so therefore should consume $1.25 \cdot 10^6 \cdot 4bytes = 5 \cdot 10^6 bytes$ or about 5 MB like the sc array. The results of the heap dump analysis are shown in figure 29. MPI additionally allocates some data for its communication, which in this case adds up to 5.7 MB, but this is not managed by the implementation itself so could possibly be reduced by issuing less MPI calls or doing so more efficiently in terms of memory usage. The absolute memory footprint of these MPI objects is however small compared to overall memory usage and attempting to reduce it would possibly reduce communication performance so it is not recommended to attempt to reduce this.

In order to ensure discrepancies can be accounted for, and to see if there are scaling effects as the number of particles increases, another benchmark is run on the same setup but with 10^6 total particles in the benchmark. This should produce a memory footprint that is exactly 10% of the previous one as there are 1/10 the number of particles, excluding the MPI objects. The dump after communication is shown in figure 31 and should be compared to figure 29.

There are some discrepancies in memory footprint between the actual usage and theoretical usage, both within the memory dedicated by sc as well as that in the std::vectors. The discrepancies in figure 31 are very small, the vector allocated memory should be about 6.5 MB large but actual consumption is 6.7 MB while the sc allocated memory is the exact expected size of 0.5 MB. For the larger benchmark shown in figure 29 the memory footprint exceeds the predicted values as well, allocating 66.7 MB for the vectors compared to the expected 65 MB and 8 MB for the sc array compared to the expected 5 MB. The vector allocated memory is ten times as large for the ten times larger particle number, within the rounding of the smaller 6.7 MB figure, which follows expectation. There are clearly some elements here which allocate additional memory but this discrepancy only accounts for a 2.6% increase in memory consumption which is well within any overhead required for managing the vectors and should not cause any problems. The sc array however produces a footprint which is 60% larger than it should be, which is concerning for the scaling of this solution to much larger problem sizes.

For the step of linking the particles to the quadrants which contain them, a heap dump is produced after the linking which is shown in figures 30 and 32. The discrepancy to figures 29 and 31 are only in the addition of the allocation stemming from the *p8est_search_all* function call. For the 10^7 particles case this is 8 MB, which is larger than the expected value of 5 MB. In the 10^6 particles case this added allocation is 0.5 MB matching the prediction exactly.

Both this vector stored on each quadrant and the sc array are supposed to allocate 5 MB of storage but instead allocate 8 MB in the case of 10^7 particles. This makes sense if one considers that the sc array is what is passed to the search function, so if extra elements were to be allocated in this array they would be present in the search function as indices which have no particle stored in the actual particle data storage. There is however no explanation for why this would be happening as the sc array shows the correct number of elements, the same as the number of particles in the particle data storage, when analysing them in runtime. This phenomena also only occurs in this larger particle size example and not in the smaller one, so further exploration may be required.

This shows that the particle data storage is very efficiently allocated with very little overhead, however the allocation of the sc array and the vectors within the quadrant data opens some questions into the efficiency with which memory was allocated.

Figures 29, 30, 31 and 32 show a breakdown of the memory heap at different times in the benchmark, generated using gperftools. In the top left of each figure is the total memory allocated, the total memory shown in the figure, and which nodes/edges were dropped in the figure. A node in this figure represents a function call, the size of which represents how much memory is allocated within that call. Each node, such as the very top *_start*, shows the memory allocated directly by it as well as the memory allocated by its children. An edge connecting a parent node to a child node shows how much memory is allocated by the respective child node. Percentages in a node show the percentage of total memory the number represents.

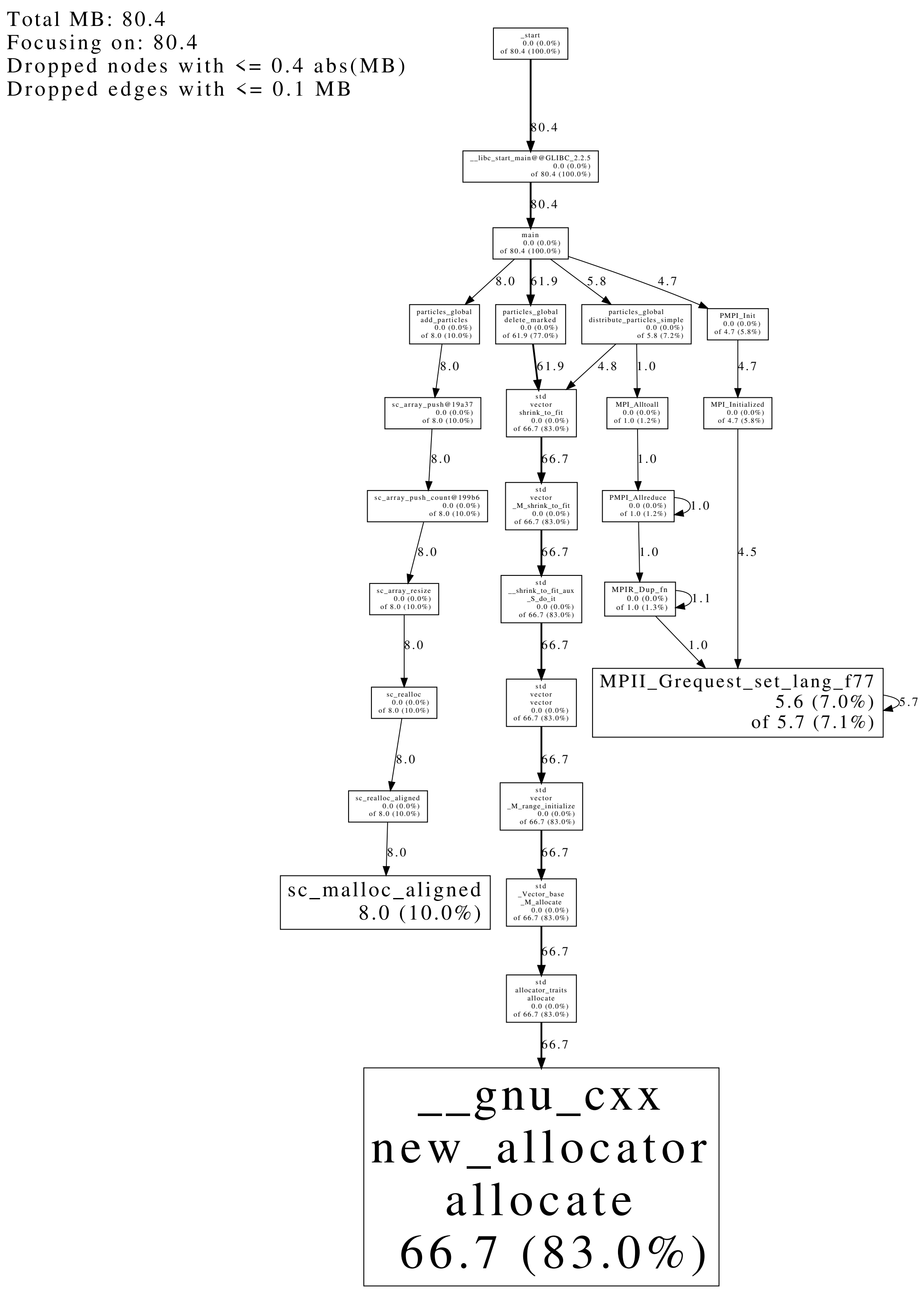

Total MB: 80.4
Focusing on: 80.4
Dropped nodes with <= 0.4 abs(MB)
Dropped edges with <= 0.1 MB
_start 0.0 (0.0%) of 80.4 (100.0%)
80.4
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 80.4 (100.0%)
80.4
main 0.0 (0.0%) of 80.4 (100.0%)
8.0
61.9
5.8
4.7
particles_global add_particles 0.0 (0.0%) of 8.0 (10.0%)
particles_global delete_marked 0.0 (0.0%) of 61.9 (77.0%)
particles_global distribute_particles_simple 0.0 (0.0%) of 5.8 (7.2%)
PMPI_Init 0.0 (0.0%) of 4.7 (5.8%)
8.0
61.9
4.8
1.0
4.7
sc_array_push@19a37 0.0 (0.0%) of 8.0 (10.0%)
std vector shrink_to_fit 0.0 (0.0%) of 66.7 (83.0%)
MPI_Alltoall 0.0 (0.0%) of 1.0 (1.2%)
MPI_Initialized 0.0 (0.0%) of 4.7 (5.8%)
8.0
66.7
1.0
sc_array_push_count@199b6 0.0 (0.0%) of 8.0 (10.0%)
std vector _M_shrink_to_fit 0.0 (0.0%) of 66.7 (83.0%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (1.2%)
1.0
8.0
66.7
1.0
4.5
sc_array_resize 0.0 (0.0%) of 8.0 (10.0%)
std __shrink_to_fit_aux _S_do_it 0.0 (0.0%) of 66.7 (83.0%)
MPIR_Dup_fn 0.0 (0.0%) of 1.0 (1.3%)
1.1
8.0
66.7
1.0
sc_realloc 0.0 (0.0%) of 8.0 (10.0%)
std vector vector 0.0 (0.0%) of 66.7 (83.0%)
MPII_Grequest_set_lang_f77 5.6 (7.0%) of 5.7 (7.1%)
5.7
8.0
66.7
sc_realloc_aligned 0.0 (0.0%) of 8.0 (10.0%)
std vector _M_range_initialize 0.0 (0.0%) of 66.7 (83.0%)
8.0
66.7
sc_malloc_aligned 8.0 (10.0%)
std _Vector_base _M_allocate 0.0 (0.0%) of 66.7 (83.0%)
66.7
std allocator_traits allocate 0.0 (0.0%) of 66.7 (83.0%)
66.7
__gnu_cxx new_allocator allocate 66.7 (83.0%)

Figure 29: Memory heap dump of process 0 after communication of particles, 10^7 particles overall, 8 processes, 8 quadrants.

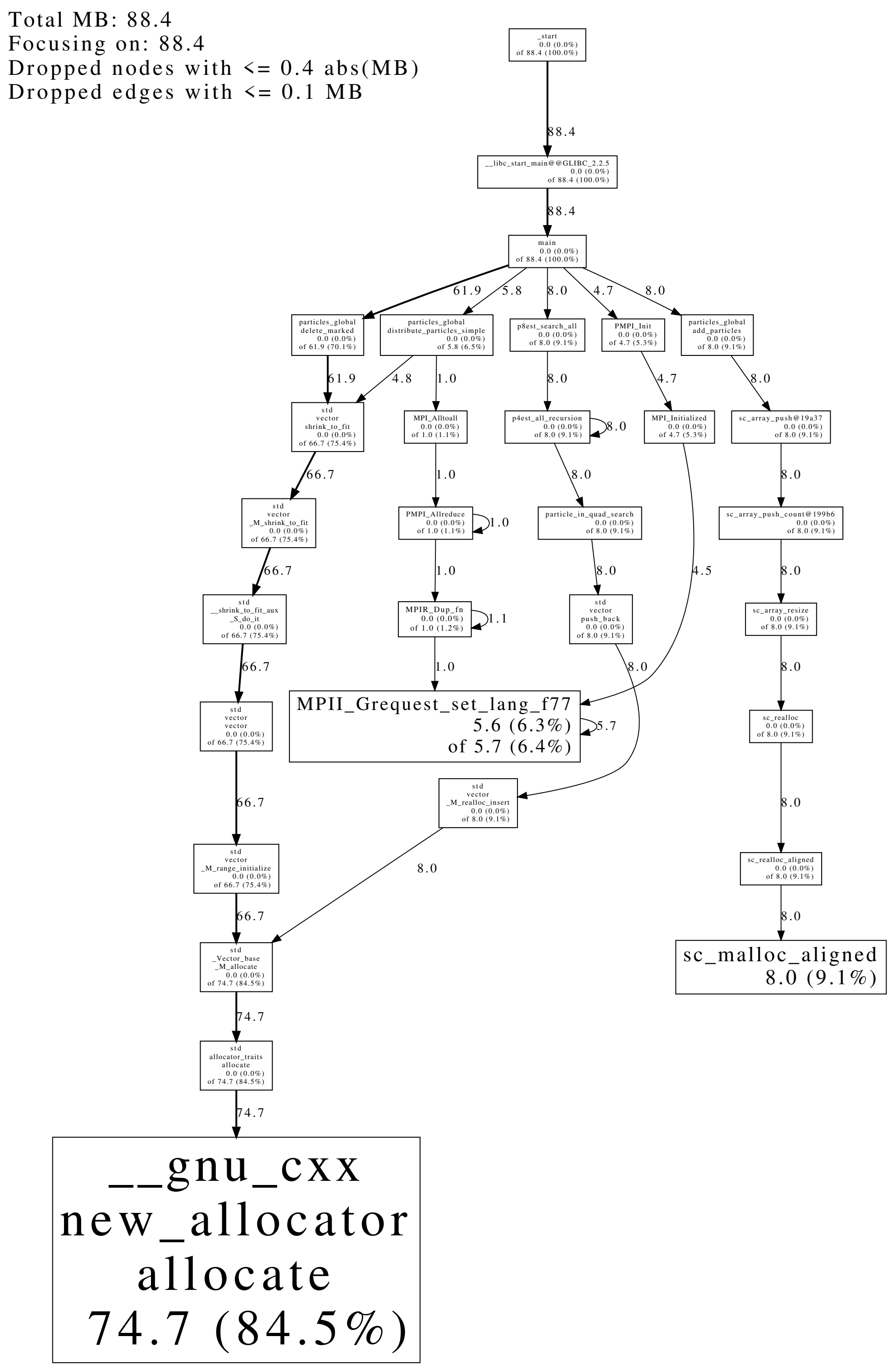

Total MB: 88.4
Focusing on: 88.4
Dropped nodes with <= 0.4 abs(MB)
Dropped edges with <= 0.1 MB
_start 0.0 (0.0%) of 88.4 (100.0%)
88.4
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 88.4 (100.0%)
88.4
main 0.0 (0.0%) of 88.4 (100.0%)
61.9
5.8
8.0
4.7
8.0
particles_global delete_marked 0.0 (0.0%) of 61.9 (70.1%)
particles_global distribute_particles_simple 0.0 (0.0%) of 5.8 (6.5%)
p8est_search_all 0.0 (0.0%) of 8.0 (9.1%)
PMPI_Init 0.0 (0.0%) of 4.7 (5.3%)
particles_global add_particles 0.0 (0.0%) of 8.0 (9.1%)
61.9
4.8
1.0
8.0
4.7
8.0
std vector shrink_to_fit 0.0 (0.0%) of 66.7 (75.4%)
MPI_Alltoall 0.0 (0.0%) of 1.0 (1.1%)
p4est_all_recursion 0.0 (0.0%) of 8.0 (9.1%)
8.0
MPI_Initialized 0.0 (0.0%) of 4.7 (5.3%)
sc_array_push@19a37 0.0 (0.0%) of 8.0 (9.1%)
66.7
1.0
8.0
8.0
std vector _M_shrink_to_fit 0.0 (0.0%) of 66.7 (75.4%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (1.1%)
1.0
particle_in_quad_search 0.0 (0.0%) of 8.0 (9.1%)
sc_array_push_count@199b6 0.0 (0.0%) of 8.0 (9.1%)
66.7
1.0
8.0
4.5
8.0
std __shrink_to_fit_aux _S_do_it 0.0 (0.0%) of 66.7 (75.4%)
MPIR_Dup_fn 0.0 (0.0%) of 1.0 (1.2%)
1.1
std vector push_back 0.0 (0.0%) of 8.0 (9.1%)
sc_array_resize 0.0 (0.0%) of 8.0 (9.1%)
66.7
1.0
8.0
8.0
std vector vector 0.0 (0.0%) of 66.7 (75.4%)
MPII_Grequest_set_lang_f77 5.6 (6.3%) of 5.7 (6.4%)
5.7
sc_realloc 0.0 (0.0%) of 8.0 (9.1%)
std vector _M_realloc_insert 0.0 (0.0%) of 8.0 (9.1%)
66.7
8.0
std vector _M_range_initialize 0.0 (0.0%) of 66.7 (75.4%)
8.0
sc_realloc_aligned 0.0 (0.0%) of 8.0 (9.1%)
66.7
8.0
std _Vector_base _M_allocate 0.0 (0.0%) of 74.7 (84.5%)
sc_malloc_aligned 8.0 (9.1%)
74.7
std allocator_traits allocate 0.0 (0.0%) of 74.7 (84.5%)
74.7
__gnu_cxx new_allocator allocate 74.7 (84.5%)

Figure 30: Memory heap dump of process 0 after particles have been linked to quadrants, 10^7 particles overall, 8 processes, 8 quadrants.

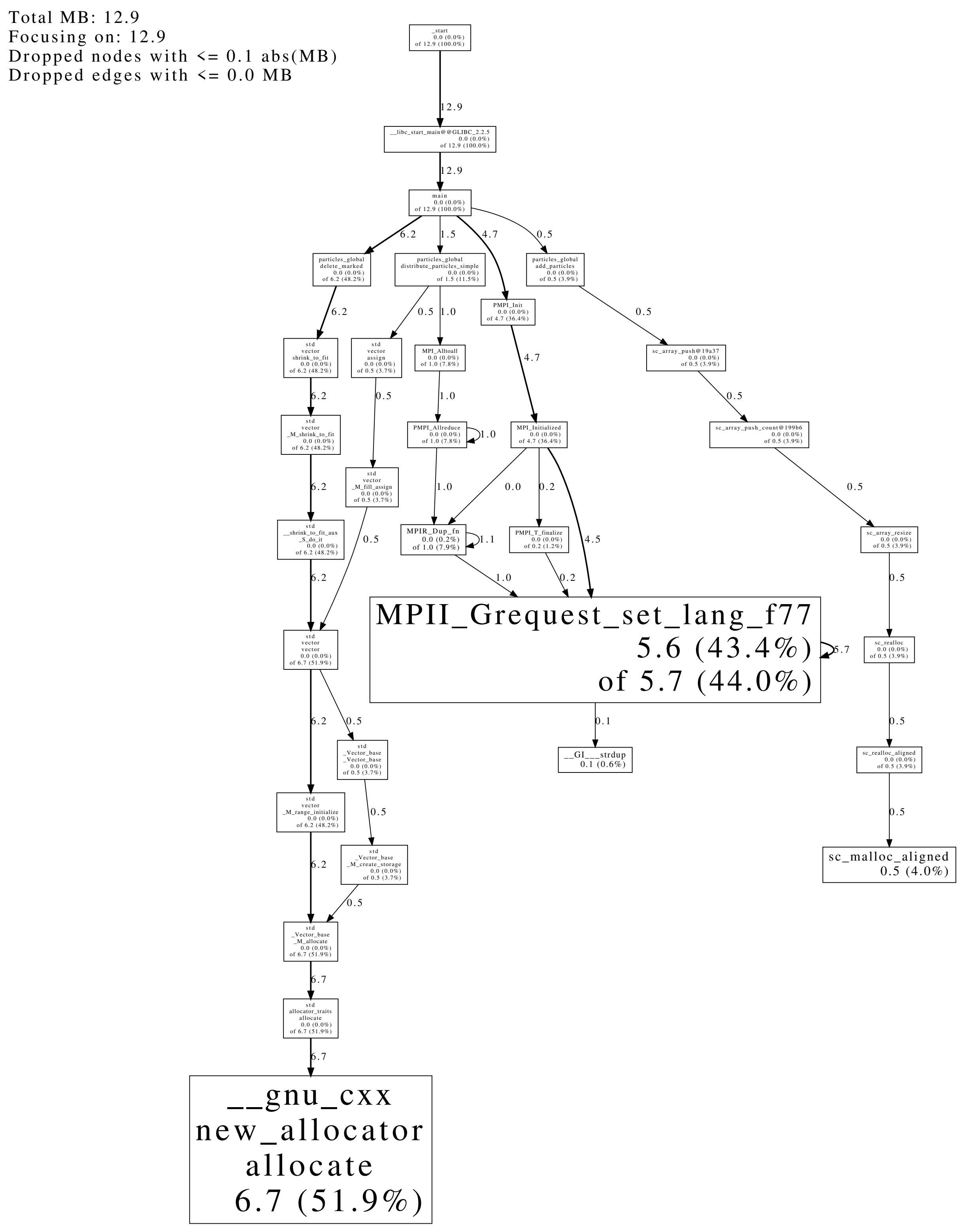

Total MB: 12.9
Focusing on: 12.9
Dropped nodes with <= 0.1 abs(MB)
Dropped edges with <= 0.0 MB
_start 0.0 (0.0%) of 12.9 (100.0%)
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 12.9 (100.0%)
main 0.0 (0.0%) of 12.9 (100.0%)
particles_global delete_marked 0.0 (0.0%) of 6.2 (48.2%)
particles_global distribute_particles_simple 0.0 (0.0%) of 1.5 (11.5%)
PMPI_Init 0.0 (0.0%) of 4.7 (36.4%)
particles_global add_particles 0.0 (0.0%) of 0.5 (3.9%)
std vector shrink_to_fit 0.0 (0.0%) of 6.2 (48.2%)
std vector assign 0.0 (0.0%) of 0.5 (3.7%)
MPI_Alltoall 0.0 (0.0%) of 1.0 (7.8%)
sc_array_push@19a37 0.0 (0.0%) of 0.5 (3.9%)
std vector _M_shrink_to_fit 0.0 (0.0%) of 6.2 (48.2%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (7.8%)
MPI_Initialized 0.0 (0.0%) of 4.7 (36.4%)
sc_array_push_count@199b6 0.0 (0.0%) of 0.5 (3.9%)
std vector _M_fill_assign 0.0 (0.0%) of 0.5 (3.7%)
__shrink_to_fit_aux _S_do_it 0.0 (0.0%) of 6.2 (48.2%)
MPIR_Dup_fn 0.0 (0.2%) of 1.0 (7.9%)
PMPI_T_finalize 0.0 (0.0%) of 0.2 (1.2%)
sc_array_resize 0.0 (0.0%) of 0.5 (3.9%)
MPII_Grequest_set_lang_f77 5.6 (43.4%) of 5.7 (44.0%)
std vector vector 0.0 (0.0%) of 6.7 (51.9%)
sc_realloc 0.0 (0.0%) of 0.5 (3.9%)
std _Vector_base _Vector_base 0.0 (0.0%) of 0.5 (3.7%)
__GI___strdup 0.1 (0.6%)
sc_realloc_aligned 0.0 (0.0%) of 0.5 (3.9%)
std vector _M_range_initialize 0.0 (0.0%) of 6.2 (48.2%)
std _Vector_base _M_create_storage 0.0 (0.0%) of 0.5 (3.7%)
sc_malloc_aligned 0.5 (4.0%)
std _Vector_base _M_allocate 0.0 (0.0%) of 6.7 (51.9%)
std allocator_traits allocate 0.0 (0.0%) of 6.7 (51.9%)
__gnu_cxx new_allocator allocate 6.7 (51.9%)
12.9
12.9
6.2
1.5
4.7
0.5
6.2
0.5
1.0
0.5
4.7
1.0
1.0
0.5
0.5
1.0
0.0
0.2
4.5
0.5
1.1
0.5
1.0
0.2
0.5
5.7
6.2
0.5
0.1
0.5
0.5
6.2
0.5
0.5
6.7
6.7

Figure 31: Memory heap dump of process 0 after communication of particles, 10^6 particles overall, 8 processes, 8 quadrants.

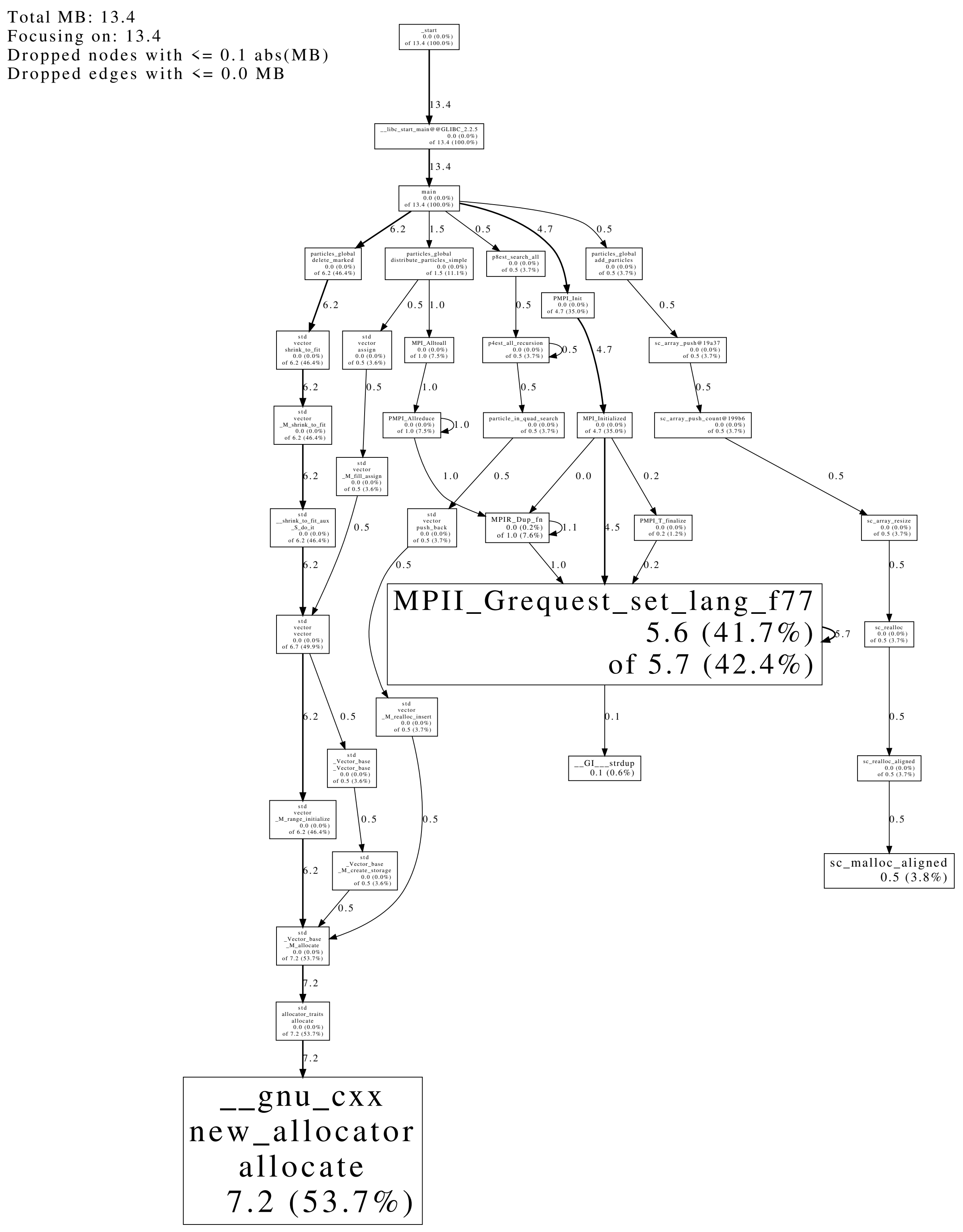

Total MB: 13.4
Focusing on: 13.4
Dropped nodes with <= 0.1 abs(MB)
Dropped edges with <= 0.0 MB
_start 0.0 (0.0%) of 13.4 (100.0%)
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 13.4 (100.0%)
main 0.0 (0.0%) of 13.4 (100.0%)
particles_global delete_marked 0.0 (0.0%) of 6.2 (46.4%)
particles_global distribute_particles_simple 0.0 (0.0%) of 1.5 (11.1%)
p8est_search_all 0.0 (0.0%) of 0.5 (3.7%)
PMPI_Init 0.0 (0.0%) of 4.7 (35.0%)
particles_global add_particles 0.0 (0.0%) of 0.5 (3.7%)
std vector shrink_to_fit 0.0 (0.0%) of 6.2 (46.4%)
std vector assign 0.0 (0.0%) of 0.5 (3.6%)
MPI_Alltoall 0.0 (0.0%) of 1.0 (7.5%)
p4est_all_recursion 0.0 (0.0%) of 0.5 (3.7%)
sc_array_push@19a37 0.0 (0.0%) of 0.5 (3.7%)
std vector _M_shrink_to_fit 0.0 (0.0%) of 6.2 (46.4%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (7.5%)
particle_in_quad_search 0.0 (0.0%) of 0.5 (3.7%)
MPI_Initialized 0.0 (0.0%) of 4.7 (35.0%)
sc_array_push_count@199b6 0.0 (0.0%) of 0.5 (3.7%)
std vector _M_fill_assign 0.0 (0.0%) of 0.5 (3.6%)
std __shrink_to_fit_aux _S_do_it 0.0 (0.0%) of 6.2 (46.4%)
std vector push_back 0.0 (0.0%) of 0.5 (3.7%)
MPIR_Dup_fn 0.0 (0.2%) of 1.0 (7.6%)
PMPI_T_finalize 0.0 (0.0%) of 0.2 (1.2%)
sc_array_resize 0.0 (0.0%) of 0.5 (3.7%)
MPII_Grequest_set_lang_f77 5.6 (41.7%) of 5.7 (42.4%)
std vector vector 0.0 (0.0%) of 6.7 (49.9%)
sc_realloc 0.0 (0.0%) of 0.5 (3.7%)
std vector _M_realloc_insert 0.0 (0.0%) of 0.5 (3.7%)
__GI___strdup 0.1 (0.6%)
std _Vector_base _Vector_base 0.0 (0.0%) of 0.5 (3.6%)
sc_realloc_aligned 0.0 (0.0%) of 0.5 (3.7%)
std vector _M_range_initialize 0.0 (0.0%) of 6.2 (46.4%)
std _Vector_base _M_create_storage 0.0 (0.0%) of 0.5 (3.6%)
sc_malloc_aligned 0.5 (3.8%)
std _Vector_base _M_allocate 0.0 (0.0%) of 7.2 (53.7%)
std allocator_traits allocate 0.0 (0.0%) of 7.2 (53.7%)
__gnu_cxx new_allocator allocate 7.2 (53.7%)

Figure 32: Memory heap dump of process 0 after particles have been linked to quadrants, 10^6 particles overall, 8 processes, 8 quadrants.

5.2 Mesh and Field Data

Benchmarking the memory consumption of p4est in a Monte Carlo particle Wigner simulation is important to find the suitability of p4est and determine what the memory footprint of a quadrant really is. For this benchmark each quadrant contains three doubles which represent field variables, a pointer to the vector which stores the particles contained in the quadrant, as well as an integer which is used to save the size of that vector as shown in snippet 5.2.

Listing 5.1: The user-defined *quadrant_data* object including field data

```
typedef struct quadrant_data{
  std::vector<int> * contained_particles;  //saves the index of all particles in the
    quadrant
  int weight;  //saves the weight of the quadrant for when the vector is deleted
  double data1;
  double data2;
  double data3;
}
quadrant_data_t;
```

P4est includes a number of other values within the quadrant data object which is managed by p4est, which are shown in snippet 5.2

Listing 5.2: The *p8est_quadrant* object

```
typedef struct p8est_quadrant
{
  p4est_qcoord_t      x, y, z;  /**< coordinates */
  int8_t              level,    /**< level of refinement */
                      pad8;     /**< padding */
  int16_t             pad16;    /**< padding */
  union p8est_quadrant_data
  {
    void                *user_data;     /**< never changed by p4est */
    long                 user_long;     /**< never changed by p4est */
    int                  user_int;      /**< never changed by p4est */
    p4est_topidx_t       which_tree;    /**< the tree containing the quadrant (used in
    auxiliary octants such as the ghost octants in p4est_ghost_t) */
    struct{
      p4est_topidx_t       which_tree;
      int                  owner_rank;
    } piggy1; /**< of ghost octants, store the tree and owner rank */
    struct{
      p4est_topidx_t       which_tree;
      p4est_topidx_t       from_tree;
    } piggy2; /**< of transformed octants, store the original tree and the target tree */
    struct{
      p4est_topidx_t       which_tree;
      p4est_locidx_t       local_num;
    } piggy3; /**< of ghost octants, store the tree and index in the owner's numbering */
  } p; /**< a union of additional data attached to a quadrant */
}
p8est_quadrant_t;
```

This is quite a number of additional variables so the quadrants can be expected to reserve more space than the particles. This *p8est_quadrant_t* object contains three structs within it, *piggy1*, *piggy2* and *piggy3*, which may be included in the quadrant or not. P4est further minimizes used storage by not allocating the *user_long* and *user_int* variables if a user defined datastructure is passed for quadrant data on initialization of p4est, so the *user_pointer* is pointing to an object.

The benchmark for this situation was chosen to be 1000 particles per process, such that the quadrants would contain at least a link to some particles within the vectors, eight processes to give an equal comparison to the particle benchmark, and 8^7 and 8^8 quadrants. These were generated from a simple cubic domain where each quadrant is refined seven or eight times. The output was again recorded just after the communication was completed, and again after the particles were linked in the quadrant data vectors. These are shown in figures 33 and 34 with 8^7 and 8^8 quadrants respectively.

The memory consumed by the data that is stored by the user within the *quadrant_data* object, which is referenced by the user pointer in the *p8est_quadrant* object is static. It can be calculated as the size of a pointer, which is eight bytes, the same size as a double, plus one integer and the three doubles stored within the quadrant data. This is a memory consumption of $8+4+3*8 = 36$ bytes per quadrant, however as all quadrants are evenly spread across processes and this is a per process memory consumption the number of quadrants needs to be divided by eight. So for the 8^7 case this would require $8^6 * 36 bytes = 9.44$ MB and $8^7 * 36 bytes = 75.5$ MB for the 8^8 case in storage just on the user defined quadrant data per process. The memory consumed by the content of the vectors is allocated by a vector allocate call not a sc_malloc therefore would not count into this calculation, and as there are a total of 1000 particles, each requiring a single integer in the vector, this would require a total of 4KB of memory and be dropped by gperftools in the visualisation anyways.

The memory heap dumps for 8^7 and 8^8 quadrants are shown in figures 33 and 34. These are again taken after the communication step, however there should be no impact when in the benchmark these snapshots are taken as the quadrant data is always allocated and does not change unless the domain is refined, coarsened or repartitioned across processes.

In the case of 8^7 quadrants, the actual memory allocated by the sc_malloc is 18.1MB compared to the 9.44MB expected to be allocated just from the user defined quadrant data. For the 8^8 quadrants case this increases to 144.7MB allocated compared to the 75.5MB expected just for user data. This scales exactly how it should be expected as $18.1MB \cdot 8 = 144.8MB$ which is within the rounding margins of the 144.7MB actually found allocated for the larger size benchmark.

The actual memory consumed by p4est per quadrant is 1.917 times larger than the calculated value in both situations. On the other side of this calculation, $\frac{144.7MB}{8^7} = 68.998$ bytes per quadrant, which can be rounded to 69 bytes per quadrant, and for the smaller benchmark $\frac{18.1MB}{8^6} = 69.046$ bytes per quadrant also rounds to 69 bytes per quadrant. This means we can confidently conclude 69 bytes of memory are allocated per quadrant when the user data contains 36 bytes, leaving 33 bytes allocated per quadrant by p4est.

P4est adds bytes each for the x, y and z coordinates (*p4est_qcoord_t* is an alias for the standard C *int32_t* type which consumes 4 bytes), one byte each for the level and *pad8* variables, 2 bytes for the *pad16* variable and 4 bytes for the *which_tree* variable (*p4est_topidx_t* is again an alias for *int32_t*) totaling 20 added bytes.

In addition, the *piggy*1, *piggy*2 and *piggy*3 structures each contain two variables each of aliased *int32_t* types which are each 4 bytes. In total this would add up to 24 bytes, which when added to the 20 bytes previously located for p4est is larger than the 33 bytes allocated by p4est per quadrant. This shows that not all of these *piggy* structures are allocated, as they are not needed in this benchmark.

This is not completely satisfactory as there is some memory allocated which is not accounted for, however the allocation is very consistent and should therefore be easy to take into account when estimating memory usage for applying p4est to a Monte Carlo particle Wigner simulation.

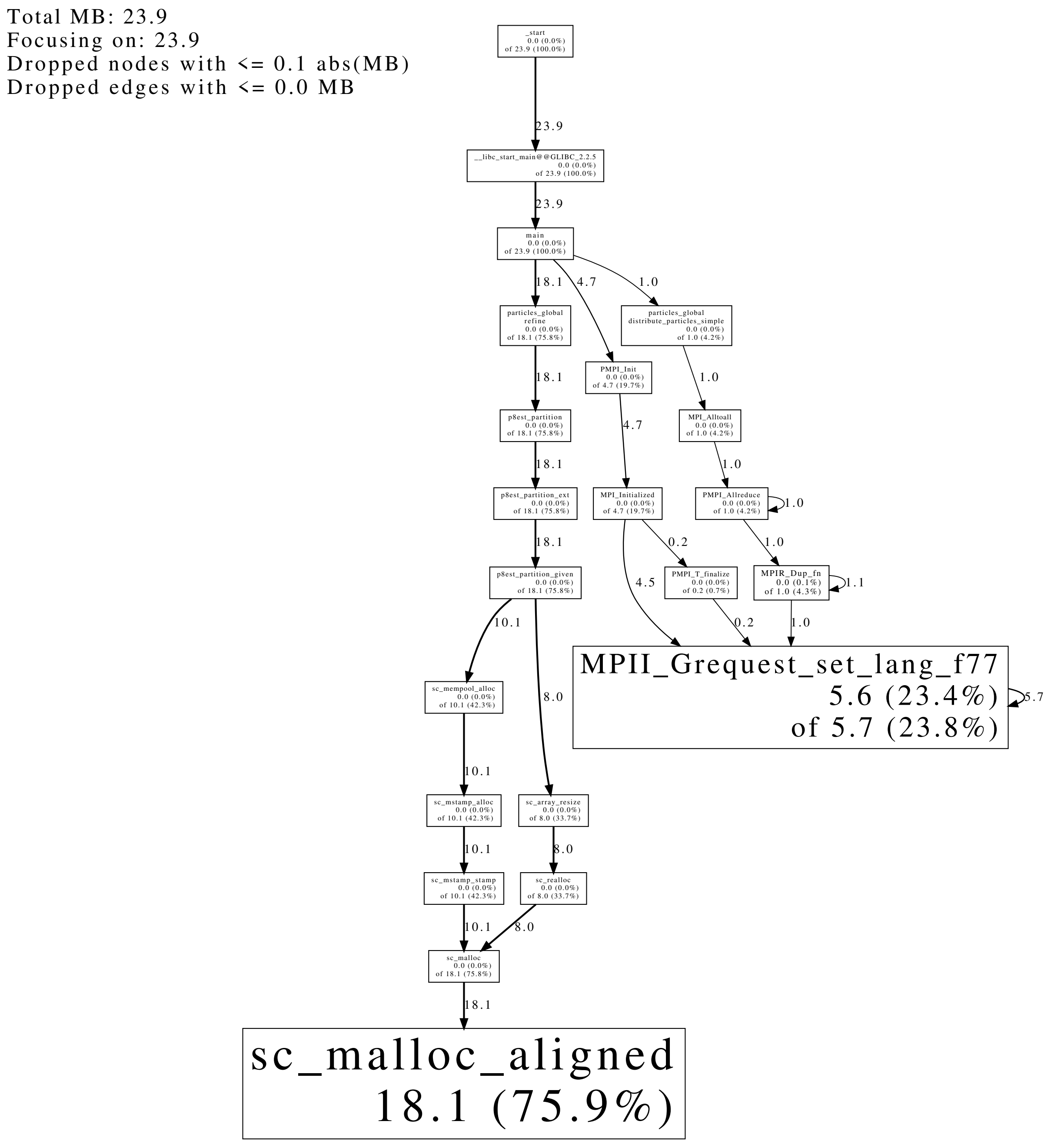

Total MB: 23.9
Focusing on: 23.9
Dropped nodes with <= 0.1 abs(MB)
Dropped edges with <= 0.0 MB
_start 0.0 (0.0%) of 23.9 (100.0%)
23.9
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 23.9 (100.0%)
23.9
main 0.0 (0.0%) of 23.9 (100.0%)
18.1
4.7
1.0
particles_global refine 0.0 (0.0%) of 18.1 (75.8%)
particles_global distribute_particles_simple 0.0 (0.0%) of 1.0 (4.2%)
PMPI_Init 0.0 (0.0%) of 4.7 (19.7%)
18.1
1.0
p8est_partition 0.0 (0.0%) of 18.1 (75.8%)
4.7
MPI_Alltoall 0.0 (0.0%) of 1.0 (4.2%)
18.1
1.0
p8est_partition_ext 0.0 (0.0%) of 18.1 (75.8%)
MPI_Initialized 0.0 (0.0%) of 4.7 (19.7%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (4.2%)
1.0
18.1
0.2
1.0
p8est_partition_given 0.0 (0.0%) of 18.1 (75.8%)
4.5
PMPI_T_finalize 0.0 (0.0%) of 0.2 (0.7%)
MPIR_Dup_fn 0.0 (0.1%) of 1.0 (4.3%)
1.1
10.1
0.2
1.0
MPII_Grequest_set_lang_f77 5.6 (23.4%) of 5.7 (23.8%)
5.7
sc_mempool_alloc 0.0 (0.0%) of 10.1 (42.3%)
8.0
10.1
sc_mstamp_alloc 0.0 (0.0%) of 10.1 (42.3%)
sc_array_resize 0.0 (0.0%) of 8.0 (33.7%)
10.1
8.0
sc_mstamp_stamp 0.0 (0.0%) of 10.1 (42.3%)
sc_realloc 0.0 (0.0%) of 8.0 (33.7%)
10.1
8.0
sc_malloc 0.0 (0.0%) of 18.1 (75.8%)
18.1
sc_malloc_aligned 18.1 (75.9%)

Figure 33: Memory heap dump post communication and particle deletion, 8^7 quadrants, 8000 particles total, 8 processes.

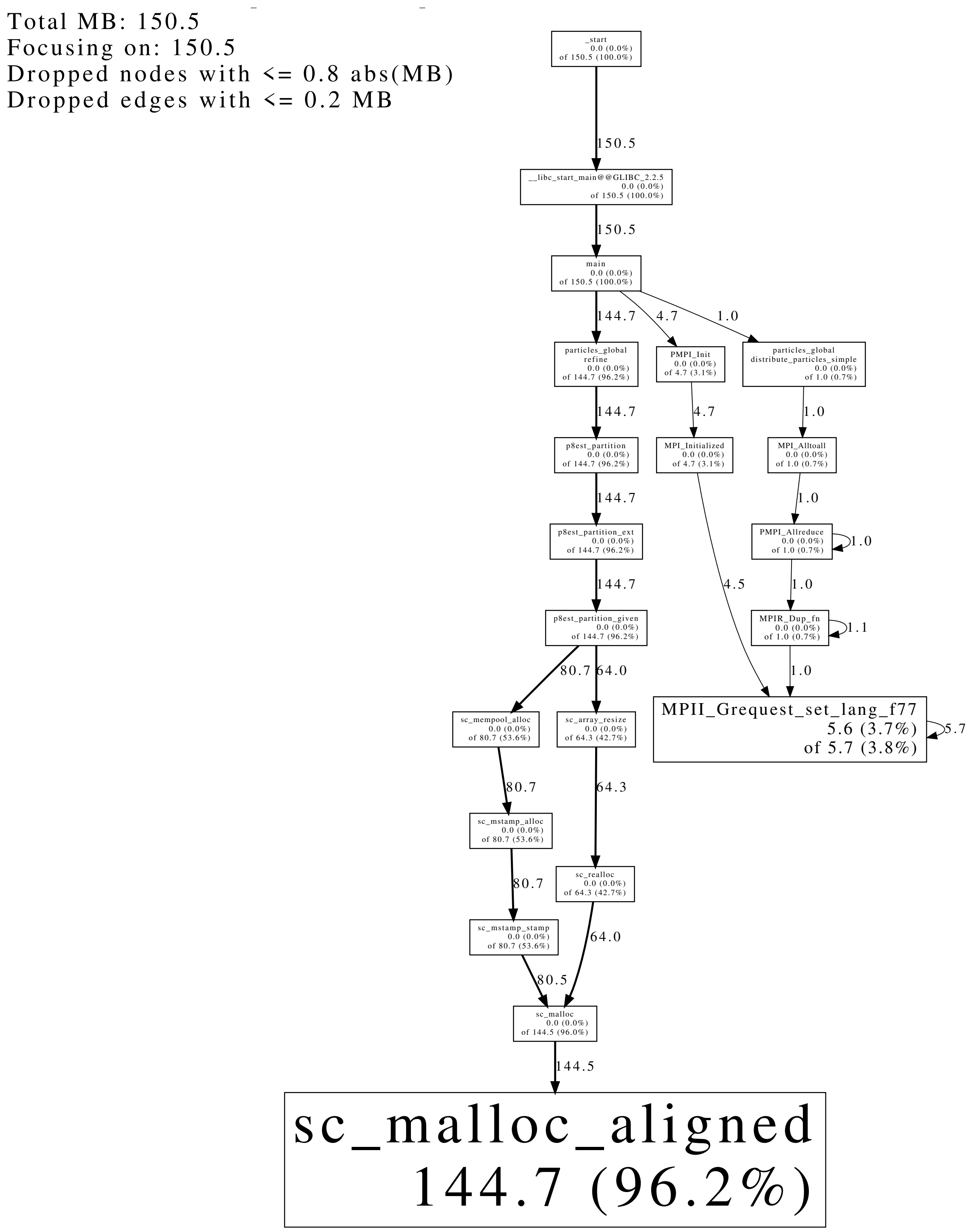

Total MB: 150.5
Focusing on: 150.5
Dropped nodes with <= 0.8 abs(MB)
Dropped edges with <= 0.2 MB
_start 0.0 (0.0%) of 150.5 (100.0%)
150.5
__libc_start_main@@GLIBC_2.2.5 0.0 (0.0%) of 150.5 (100.0%)
150.5
main 0.0 (0.0%) of 150.5 (100.0%)
144.7
4.7
1.0
particles_global refine 0.0 (0.0%) of 144.7 (96.2%)
PMPI_Init 0.0 (0.0%) of 4.7 (3.1%)
particles_global distribute_particles_simple 0.0 (0.0%) of 1.0 (0.7%)
144.7
4.7
1.0
p8est_partition 0.0 (0.0%) of 144.7 (96.2%)
MPI_Initialized 0.0 (0.0%) of 4.7 (3.1%)
MPI_Alltoall 0.0 (0.0%) of 1.0 (0.7%)
144.7
1.0
p8est_partition_ext 0.0 (0.0%) of 144.7 (96.2%)
PMPI_Allreduce 0.0 (0.0%) of 1.0 (0.7%)
1.0
144.7
4.5
1.0
p8est_partition_given 0.0 (0.0%) of 144.7 (96.2%)
MPIR_Dup_fn 0.0 (0.0%) of 1.0 (0.7%)
1.1
80.7
64.0
1.0
MPII_Grequest_set_lang_f77 5.6 (3.7%) of 5.7 (3.8%)
5.7
sc_mempool_alloc 0.0 (0.0%) of 80.7 (53.6%)
sc_array_resize 0.0 (0.0%) of 64.3 (42.7%)
80.7
64.3
sc_mstamp_alloc 0.0 (0.0%) of 80.7 (53.6%)
sc_realloc 0.0 (0.0%) of 64.3 (42.7%)
80.7
64.0
sc_mstamp_stamp 0.0 (0.0%) of 80.7 (53.6%)
80.5
sc_malloc 0.0 (0.0%) of 144.5 (96.0%)
144.5
sc_malloc_aligned 144.7 (96.2%)

Figure 34: Memory heap dump post communication and particle deletion, 8^8 quadrants, 8000 particles total, 8 processes.

6 Summary and Outlook

The objective of the thesis was to evaluate the performance of the *forest-of-octrees* datastructure as provided by p4est for potential utilization as the fundamental mesh data structure for ViennaWD. A benchmark library was developed which made possible a detailed performance evaluation of p4est in this context, using a minimal working example featuring key algorithmic aspects representative of the particle solver algorithm used in ViennaWD. A number of design choices of the benchmark library were explored and then benchmarked against each other. Their advantages and disadvantages were observed and recorded to allow the overall benchmarking library to operate as efficiently as possible. A number of bottlenecks were found which significantly slowed the execution of the code, including when particles are distributed across processes very unevenly, leading to load imbalance, and a communication bottleneck when communication loads exceeded a certain measure which is likely specific to the VSC. When avoiding these bottlenecks however, a very strong scaling of the performance when increasing the number of particles, quadrants, and processes within the benchmark was found. The strong scaling especially found excellent performance characteristics, although weak scaling also showed good performance scaling. Collecting these results, the library provides a strong framework for the future iterations of the ViennaWD simulator.

Generally p4est was found to perform very well in a variety of circumstances and provide most functionality required for the Monte Carlo particle Wigner simulation directly, although there are some features which would be beneficial which need to be further explored outside this thesis. The scaling of runtime performance was found to be excellent, with bottlenecks found not in the p4est library itself but the overall load placed on the benchmark when executed on the VSC supercomputer. The inclusion of more quadrants in the benchmark scaled very well up to a certain point at which the functions operating on quadrants took over the majority of the runtime. This should be considered not an issue, however, as this transition point is at minimum 10^6 quadrants for only 10^3 particles, likely further as the benchmark calculations would slow down the other calculations and the quadrant functions would not dominate the runtime. This is a very high number of quadrants, reaching the number of particles included in the benchmark, and a mesh which has one cell for every particle in the particle Wigner simulation is much too refined.

Both strong and weak scaling was found to be quite strong if the system is in a load balanced state, loosing no performance up to the process count available within one node and only slowing down once communication must cross the links between nodes, as should be expected. In a non-load balanced state both weak and strong scaling were quite poor, as expected, so this should be taken into account when designing a simulator based on p4est.

The longest compute times for this benchmark, in order of length, were the functions which determined the particle-cell relationships, the communication of particles across processes, and the functions which finds the processes on which a particle resides after generation or movement of the particles.

Future work should investigate the following four aspects:

First, a problem within the refinement functions using data stored within the quadrants themselves, not generated by some calculation over the field based on coordinates, is that it makes recursive refinement impossible. Every time a new quadrant is generated p4est automatically fills variables that it manages while all user defined ones are left empty until they are filled. It would be advantageous to inherit some data from the parent quadrant of the newly generated ones, but this was not part of the scope of this benchmark. An init function can be passed to the refine function, which is called on every new quadrant, so developing an algorithm which finds the parent should allow this inheritance to be made possible.

Second, as previously discussed, the Wigner function describes a density distribution function. Particles are then generated according to this density distribution function for the domain which is used in the particle Wigner simulation. An investigation into the best method to execute such a particle generation event, generating particles in a master-worker configuration, only generating locally on each process, or doing it distributed across all processes as in the benchmarking in this thesis, would likely allow for some further optimization. Similarly, when particle generation events place a very uneven load across all processes a way to refine the mesh using this particle distribution function and its defined variability across space instead of the real particle distribution, which would need the particle-cell search to be completed before it can be used in refinement, would likely allow for even more performance to be achieved.

Third, although two different memory layouts were benchmarked in this thesis and there was a slight advantage to the SoA memory layout, there was no clear conclusion drawn from the performance of the two. The study conducted in [24] also found the SoA memory layout to be the better choice, so this is likely to be optimal, but may differ in the final implementation. When executing a real particle Wigner simulation using differing geometries there may be more significant performance differences between the two options, therefore a further study comparing the performance of the SoA and AoS layouts using such a real world, complicated geometry may be advantageous.

Fourth, the choice of compilers and MPI libraries for this benchmarking was done without much testing or benchmarking across options; therefore, the performance could possibly be improved by selecting a different combination of compilers and/or MPI libraries. There was also no investigation into the advantages gained when using certain compiler flags which could optimize performance further, so more investigation into these could possibly lead to even better performance. These were however also not necessary for the benchmarking of the library here as analysing the absolute runtime performance was not the main goal; instead, the goal was to compare the performance when changing a number of parameters (particle, quadrant and process count).

References

[1] Dragica Vasileska and Stephen M Goodnick. “Computational electronics”. In: *Materials Science and Engineering: R: Reports* 38.5 (2002), pp. 181–236. DOI: https://doi.org/10.1016/S0927-796X(02)00039-6.

[2] Rajesh Venugopal et al. “Simulating quantum transport in nanoscale transistors: Real versus mode-space approaches”. In: *Journal of Applied physics* 92.7 (2002), pp. 3730–3739. DOI: https://doi.org/10.1063/1.1503165.

[3] E. Schrödinger. “Quantisierung als Eigenwertproblem”. In: *Annalen der Physik* 384.4 (1926), pp. 361–376. DOI: https://doi.org/10.1002/andp.19263840404.

[4] E. Wigner. “On the quantum correction for thermodynamic equilibrium”. In: *Physical Review* 40.5 (1932), pp. 749–759. DOI: 10.1103/PhysRev.40.749.

[5] Richard Phillips Feynman. “Space-time approach to non-relativistic quantum mechanics”. In: *Reviews of modern physics* 20.2 (1948), p. 367. DOI: 10.1103/RevModPhys.20.367.

[6] Umberto Ravaioli et al. “Investigation of ballistic transport through resonant-tunnelling quantum wells using wigner function approach”. In: *Physica B+C* 134.1 (1985), pp. 36–40. ISSN: 0378-4363. DOI: https://doi.org/10.1016/0378-4363(85)90317-1.

[7] Ivan Dimov. “Monte Carlo Algorithms for Linear Problems”. In: *Pliska Studia Mathematica Bulgarica* 13.1 (2000), pp. 57–77.

[8] L. Shifren and D.K. Ferry. “Particle Monte Carlo simulation of Wigner function tunneling”. In: *Physics Letters A* 285.3 (2001), pp. 217–221. ISSN: 0375-9601. DOI: https://doi.org/10.1016/S0375-9601(01)00344-9.

[9] Ivan Dimov Jean Michel Sellier Mihail Nedjalkov and Siegfried Selberherr. “A benchmark study of the Wigner Monte Carlo method”. In: *Monte Carlo Methods and Applications* 20.1 (2014), pp. 43–51. DOI: doi:10.1515/mcma-2013-0018.

[10] L Shifren and DK Ferry. “Particle Monte Carlo simulation of Wigner function tunneling”. In: *Physics Letters A* 285.3-4 (2001), pp. 217–221. DOI: https://doi.org/10.1016/S0375-9601(01)00344-9.

[11] P. Ellinghaus. *Two-Dimensional Wigner Monte Carlo Simulation for Time-Resolved Quan- tum Transport with Scattering*. 2016. DOI: 10.34726/hss.2016.35764.

[12] Carsten Burstedde et al. “ALPS: A framework for parallel adaptive PDE solution”. In: *Journal of Physics: Conference Series* 180.1 (May 2009), p. 012009. DOI: 10.1088/1742-6596/180/1/012009. URL: https://dx.doi.org/10.1088/1742-6596/180/1/012009.

[13] Lucas C Wilcox et al. “A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media”. In: *Journal of Computational Physics* 229.24 (2010), pp. 9373–9396.

[14] Lucas C. Wilcox Carsten Burstedde and Omar Ghattas. "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees". In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: http://dx.doi.org/10.1137/100791634.

[15] Gerhard Wellein Georg Hager. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010. ISBN: 978-1439811924.

[16] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. "Distributed shared memory: Concepts and systems". In: *IEEE Parallel & Distributed Technology: Systems & Applications* 4.2 (1996), pp. 63–71. DOI: 10.1109/88.494605.

[17] Galina Reshetova, Vladimir Cheverda, and Vitaly Koinov. "Comparative efficiency analysis of mpi blocking and non-blocking communications with coarray fortran". In: *Supercomputing: 7th Russian Supercomputing Days, RuSCDays 2021, Moscow, Russia, September 27–28, 2021, Revised Selected Papers 7*. Springer. 2021, pp. 322–336.

[18] G.M. Shipman et al. "Infiniband scalability in Open MPI". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. 2006, 10 pp.-. DOI: 10.1109/IPDPS.2006.1639335.

[19] Diah Ayu Retnani Wulandari and Mochamad Edoward Ramadhan. "Performance comparison analysis library communication cluster system using merge sort". In: *Journal of Physics: Conference Series*. Vol. 1008. 1. IOP Publishing. 2018, p. 012002. DOI: 10.1088/1742-6596/1008/1/012002.

[20] Jin Nengzhi et al. "Comparative Research on High-Speed Networks of High Performance Computing Cluster Based on MPIGRAPH". In: *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*. 2020, pp. 579–583. DOI: 10.1109/ICCC51575.2020.9344976.

[21] *VSC-4 - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path*. https://www.top500.org/system/179697/. Accessed: 2024-02-10.

[22] *VSC-5 - MEGWARE SLIDESX, AMD EPYC 7713 64C 2GHz, Infiniband HDR*. https://www.top500.org/system/180056/. Accessed: 2024-02-10.

[23] Brice Goglin. "High Throughput Intra-Node MPI Communication with Open-MX". In: *17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009)*. Ed. by IEEE. Weimar, Germany, Feb. 2009. DOI: 10.1109/PDP.2009.20. URL: https://inria.hal.science/inria-00331209.

[24] Alexander Adel. *Dynamic Particle Data Structures for Wigner Monte Carlo Simulations*. 2023. DOI: 10.34726/hss.2023.107246.

[25] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533. DOI: https://dl.acm.org/doi/10.1145/42411.42415.